Scaling Up The Performance of Distributed Key-Value Stores Using Emerging Technologies for Big Data Applications

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Hebatalla Eldakiky

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Prof. David Hung-Chang Du

August, 2021

Acknowledgements

This thesis would not be possible without the kind help and support of many individuals around me. I would like to express my gratitude and sincere thanks to all of them.

First and foremost, I would like to thank my advisor, Prof. David H.C. Du, for his continuous guidance and support of my research. He has been also an excellent role model for critical thinking, hard work, great passion, and commitment toward research. I learned from him how to identify valuable research issues and attack them directly. I would not have become the researcher that I am today without his invaluable lessons and assistance. His mentorship invokes my tremendous interest in doing research, and will have long lasting impact on my future career.

I also would like to extend my gratitude to Prof. Zhi-Li Zhang, Prof. Jon Weissman, Prof. Chris Kim, Prof. David Lilja and Prof. Haiyi Zhu for serving on my thesis and preliminary examination committees and for their invaluable comments and suggestions.

I will forever be indebted to several persons who have supported me to a great extent during my PhD. My first supporters are my parents who gave me all the opportunities and experiences that have made me who I am. Their prayers, encouragement and help give me strength to overcome all challenges during my PhD journey. Very special thanks goes to my husband Ibrahim, for his ongoing support and for being my closest friend in pursuing our PhD career together. I am also a proud mother of two amazing Kids, Eyad and Adhem, who are the joy of my life and made me more fulfilled than I could have ever imagined. I would like also to extend my gratitude to my lovely sister Basma, my niece Malak and my nephew Asser, for their support and encouragement regardless the remote distance between us. I am also thankful to my lifelong friends Nehal and Dalia for their continuous support.

I would like also to thank my second family here in the USA who have supported me throughout my PhD journey and made me feel like being at home, Specially Eman Ramadan, Sara Aktham, Ahmed Zamzam, Reem Ali, Amr Magdy, Adel Elmahdy, Ebtehal Bahnasy, and Sara Morsy.

I was fortunate enough to be a member of the Center for Research in Intelligent Storage (CRIS) and I would like to thank the many former and current members of it: Zhichao Cao, Fenggang Wu, Baoquan Zhang, Bingzhe Li, Ming-Hong Yang, Chai-Wen Hsieh, Jim Diehl, Ziqi Fan, Manas Minglani, Xiang Cao, Nae Young Song, Yixun Wei, Huibing Dong, Wenlong Wang and Haoyu Gong. I sincerely appreciate their help and support that greatly enriched my knowledge and my research capabilities.

Last but not least, I would like to thank the University of Minnesota, the National Science Foundation (NSF) I/UCRC Center Research in Intelligent Storage and the following NSF awards 1439662, 1525617, 1536447, 1708886, 1763008, and 1812537 for supporting my research.

Dedication

To those who held me up over the years, my dad Mohamed, my mom Sanaa. To my husband Ibrahim, my lovely sister Basma, my kids Eyad, Adhem, Malak and Asser, and all other family members and friends.

Abstract

The explosion in the amount of data with the development of internet and cloud computing prompted much research to develop systems that are able to store and process this data efficiently. As data is generated by different sources with un-unified structures, NoSQL databases emerged as a solution due to their flexibility and high performance. Key-value stores, one of the NoSQL databases categories, are widely used in many big data applications. This wide usage is for its efficiency in handling data in key-value format, and flexibility to scale out without significant database redesign.

In key-value stores, with such huge amount of data, data can not be stored in a single storage server. Thus, this data has to be partitioned across multiple storage instances. Key-value queries have to access the information of these partitions to locate the target key-value pairs, and be directed to the right storage node that physically holds the data. This scenario introduces further forwarding steps in the path to the target storage node. These additional forwarding steps affect the query response time.

Recently, the power and flexibility of software-defined networks with the evolution of the programmable switches lead to a programmable network infrastructure where in-network computation can help accelerating the performance of applications. This can be achieved by offloading some computational tasks to the network to improve data access performance when applications access storage through network. However, what kind of computational tasks should be delegated to the network to accelerate applications performance? To solve the partition management problem in key-value stores, we developed TurboKV, an in-switch coordination model, which utilizes the programmable switches as partition management nodes and monitoring stations to scale up the performance of the distributed key-value stores. Our in-switch coordination model removes the load of routing the requests from storage nodes without introducing any additional forwarding steps in the path to the target storage node.

Moreover, some key-value stores omit the transaction concepts because of their effect on the scalability and decreasing the performance of key-value stores, which are the key targets of any existing key-value store system. This effect is due to the complexity, locking, starvation introduced by transactions and the interference with the non-transaction

operations. In order to provide an efficient support for the transactions in key-value stores, we propose TransKV, an extension to our first work TurboKV, which introduces a networking support for transaction processing in distributed key-value stores. TransKV utilizes the programmable switches as a transaction coordinator who can decide whether the transaction can proceed to be processed by the storage nodes or just aborted from the network.

On the storage node side, Seagate developed a new drive called "Kinetic drive". The Kinetic drive is an independent active disk accessible by Ethernet connection. This enables applications to directly connect to the drive via IP address, and retrieve a piece of data. Kinetic drive can also carry out key-value pair operations on its own. So in large scale data management, a set of Kinetic drives can be used to exploit parallelism in satisfying user requests, and solve the bottleneck caused by queuing of requests in the storage server which manages multiple HDDs/SDDs. On the other hand, Kinetic drive has a limited bandwidth and capacity. Therefore, a careful allocation scheme is needed to allocate key-value pairs to a set of Kinetic drives taking into account each drive's limited bandwidth and capacity. To this extent, we developed a key-value pair allocation strategy for Kinetic drives. This strategy takes into consideration the data popularity, the limited capacity and the bandwidth of Kinetic drive to avoid queuing on the level of the drive.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Through the massive use of mobile devices, data clouds, and the rise of Internet of Things [6], enormous amount of data has been generated and analyzed for the benefit of society at a large scale. The amount of this generated data is extremely growing at a rapid rate. According to EMC Digital Universe with Research & Analysis by IDC [7], the digital universe is doubling in size every two years, and by 2020 the digital universe will reach around 44 zettabytes or 44 trillion gigabytes of data. Consequently, there is a deed need for efficient tools, which can achieve high performance in storing and processing these large amounts of data. In other words, data management and processing become essential in the big data research.

Most of nowadays data is generated by different sources with un-unified structures. This data can be text, image, audio, video, etc. Hence, this data is often maintained in key-value store, which is widely used due to its efficiency in handling data in key-value format, and flexibility to scale out without significant database redesign. In key-value stores, object is treated as an opaque collection. Each object is represented by two attributes: key and value. The key is used as an unique identifier to store, read, modify or delete the record. The value is a variable-length object, that can be used to store any type of data. Examples of popular key-value stores include Amazon's Dynamo [8], Redis [9], RAMCloud [10], LevelDB [11] and RocksDB [12].

On the networking side, Software-Defined Network (SDN) simplifies network devices by introducing a logically centralized controller (control plane) to manage simple programmable switches (data plane). SDN controllers set up forwarding rules at the

programmable switches and collect their statistics using OpenFlow APIs [13]. As a result, SDN enables efficient and fine-grained network management and monitoring in addition to allowing independent evolution of the controller and the programmable switches. Recently, the Programming Protocol-Independent Packet Processor (P4) [14] unleashes capabilities that give the freedom to create intelligent network nodes performing various functions. Thus, applications can boost their performance by offloading part of their computational tasks to these programmable switches to be executed in the network. Nowadays, programmable networks get a bigger foot in the data center doors. Google cloud started to use the programmable switches with P4Runtime [15] to build and control their smart networks [16]. Some Internet Service Providers (ISPs), such as AT&T, have already integrated programmable switches in their networks [17]. These switches can be controlled by network operators to adapt to the current network state and application requirements, to provide more flexibility and high throughput [18, 19].

Recently, there is an uptake in leveraging programmable switches to improve distributed systems, e.g., NetPaxos [20, 21], NetCache [22], NetChain [23], DistCache [24] and iSwitch [25]. This uptake is due to the massive evolution on the capailities of these network switches, e.g., Tofino ASIC from Intel [26]. This ASIC provides submicrosecond per-packet processing delay with bandwidth up to 6.5 Tbs and throughput of few billions of packets processed per second. There is also the second generation, Tofino2 [27], with bandwidth up to 12.8 Tbs. These systems use the switches to provide orders of magnitude higher throughput than the traditional server-based solutions.

In parallel to the evolution on network devices, the Object Storage Devices (OSD) and active disks were introduced. These devices can manipulate data in-terms of objects instead of file blocks. The Kinetic drive is an example of an OSD and active disks introduced by SeaGate. The Kinetic drive has its own CPU and RAM with built-in LevelDB [11]. It can perform the basic key-value pair operations efficiently without going through different stacked layers of software and hardware introduced by the storage server which manage block based disk drives. Through the Ethernet connection, enabled in the Kinetic drive, applications can request any data they need by only connecting to the suitable Kinetic drive, and data will be transferred over this connection. So we can say that the Kinetic drive is an independent small key-value storage.

Now, as we have control over both network and storage, it is time to think about

how to improve data access performance when applications access storage through network. In our work, we focus on improving the performance of accessing data from key-value stores. In key-value stores with such huge amount of data, this data can not be stored in a single storage server. Thus, it has to be partitioned across different storage instances. In order to locate a piece of data, data paritions and their storage node mapping (directory information) are either stored on a single coordinator node, e.g., the master coordinator in distributed Google file system [28], or replicated over all storage instances [8, 9, 29]. These approaches increase queries' response time by introducing additional forwarding steps in the path between the application and the target storage nodes. In the first piece of our work, we developed a novel distributed key-value store architecture that leverages the power and flexibility of the new generation of programmable switches to scale up the performance of the distributed key-value stores. Our approach offloads the partitions management and query routing to be carried out in network switches. It uses in-switch coordination model which utilizes the programmable switches as: 1) partition management nodes to store and manage the directory information of key-value store; and 2) monitoring stations to measure the load of storage nodes, where this monitoring information is used to balance the load among storage nodes. Because requests already pass by network switches in their way to the storage nodes, our in-switch coordination model removes the load of routing the requests from storage nodes without introducing any additional forwarding steps in the request path to the target storage node.

Moreover some key-value stores [29, 30] omit the transaction concepts because of their effect on the scalability and decreasing the performance of key-value stores, which are the key targets of any existing key-value store system. This effect is due to the complexity, locking, starvation introduced by transactions and the interference with the non-transaction operations. However, other key-value stores [8, 9] support transaction concepts by introducing the notion of transaction coordinator, that is responsible for aborting or accepting the transaction. This transaction coordinator introduces further forwarding steps in the processing of key-value queries. To further extend our work with the programmable switches in improving the performance of key-value stores, in our second piece of work, we are proposing a network support for transaction processing using the programmable switches. We believe that network latency has a significant impact

on the performance of transactions which have to be processed by the storage system in order to ensure serializability. Our approach utilizes the programmable switches to execute the transactions processing logic in the network. We are developing a variation of the Time Stamp Ordering algorithm (TSO) [31] on the programmable switches. If a transaction can be pushed to processing according to the TSO logic, it is accepted and forwarded to the target storage nodes to start processing. Otherwise, the transaction is aborted early by the programmable switches, and packets are routed back to the client.

On the storage server side, the traditional key-value storage implementation is on storage servers which manage block-based disk drives. This implementation consists of multiple layers of software and hardware stacked together in order to enable a data path between two poorly compatible systems: an object-oriented application layer and a hardware layer (HDDs, SSDs and tape). When many users share access to a business application, all requests are sent to the storage server and may build up in the queue. The response time for each I/O starts to increase, which decreases the server throughput and leads to a performance bottleneck [32]. By taking the advantage of the Kinetic drive as being an independent small key-value store, exploiting parallelism using multiple Kinetic drives removes the server bottleneck and improves system performance [33, 34]. However, each Kinetic drive has limited bandwidth and limited capacity. It can only hold data up to its capacity, and support data access rate up to its bandwidth. In our third piece of work, we developed a key-value pairs placement scheme which takes the limited bandwidth and capacity of the drive as factors in the data allocation process. This scheme allocates key-value pairs to the Kinetic drive which has enough space to them, and also can satisfy their search requests according to its bandwidth. Our scheme aims at resolving the disk bandwidth and capacity issue to avoid the queuing at each drive, and hence avoid the performance bottleneck issue on the level of each drive.

The rest of this thesis is organized as follows. Chapter 2 provides the background about emerging technologies. Chapter 3 focuses on the details of TurboKV, our in-switch coordination approach to scale up the performance of distributed key-value stores. Chapter 4 focuses on the details of TransKV, which is a networking support for transaction processing in distributed key-value stores. Chapter 5 discusses our key-value pairs allocation strategy for Kinetic drives based key-value stores. Finally, our work will be concluded in Chapter 6.

Chapter 2

Emerging Hardware Technologies

2.1 Programmable Switches

Software-Defined Network (SDN) simplifies network devices. It enables efficient and fine-grained network management and monitoring in addition to allowing independent evolution of the controller and programmable switches. Recently, Programming Protocol-independent Packet Processors (P4) [14] have been introduced to enrich the capabilities of network devices by allowing developers to define their own packet formats and build the processing graphs of these customized packets. P4 is a programming language designed to program parsing and processing of user-defined packets using a set of match/action tables. It is a target-independent language, thus a P4 compiler is required to translate P4 programs into target-dependent switch configurations.

Figure 2.1(a) represents the five main data plane components for most of modern switch ASICs. These components include programmable parser, ingress pipeline, traffic manager, egress pipeline and programmable deparser. When a packet is first received by one of the ingress ports, it goes through the programmable parser. The programmable parser, as shown in Figure 2.1(a), is modeled as a simple deterministic state machine, that consumes packet data and identifies headers that will be recognized by the data plane program. It makes transitions between states typically by looking at specific fields in the previously identified headers. For example, after parsing the Ethernet header in the packet, the next state will be determined based on the Ethertype field, whether it will be reading a VLAN tag, an IPv4, an IPv6 or a customized header.

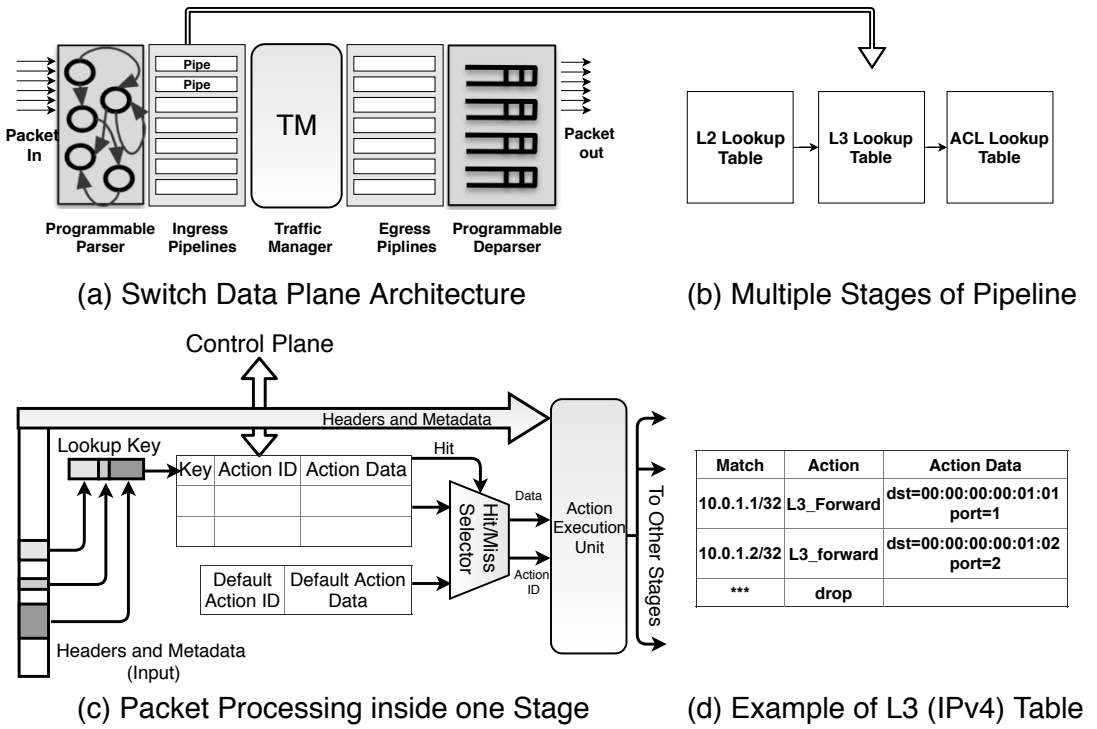


Figure 2.1: Primelinries on Programmable Switches

After the packet is processed by the parser and decomposed into different headers, it goes through one of the ingress pipes to enter a set of match-action processing stages as shown in Figure 2.1(b). In each stage, a key is formed from the extracted data and matched against a table that contains some entries. These entries can be added and removed through the control plane. If there is a match with one of the entries, the corresponding action will be executed, otherwise a default action will be executed. After the action execution, the packet with all the updated headers from this stage enters the next stages in the pipeline. Figure 2.1(c) illustrates the processing inside a pipeline stage, while Figure 2.1(d) shows an example IPv4 table that picks an egress port based on destination IP address. The last rule drops all packets that do not match any of the IP prefixes, where this rules corresponds to the default action.

After the packet finishes all the stages in the ingress pipeline, it is queued and switched by the traffic manager for an egress pipe for further processing. At the end, there is a programmable deparser that performs the reverse operation of the parser. It reassembles the packet back with the updated headers with the same order defined the

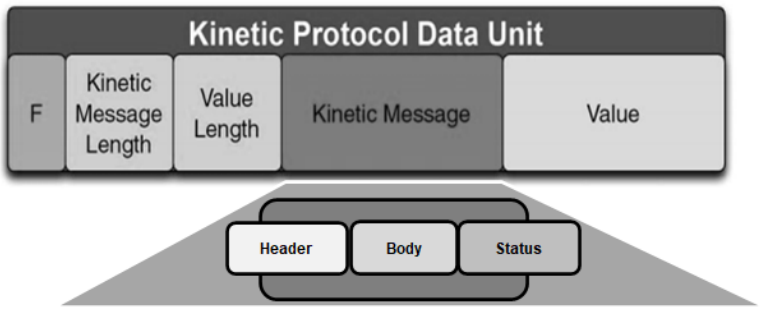


Figure 2.2: Kinetic Protocol Data Unit Message Packet [4]

original packet so that the packet can be sent back onto the wire through an egress port to the next hop in the path to its final destination.

2.2 Kinetic Drives Preliminaries

Kinetic drive is a key-value storage active device developed by SeaGate. It is accessible by an Ethernet connection. It has an open application programming interface (API) and associated libraries that enable applications to connnect to the target storage device directly without passing multiple legacy layers in the traditional storage systems. They enable applications to manipulate objects and manage clusters while letting the drive control some functionalities. These functionalities include managing key ordering, handling device to device data migration with peer-to-peer data copy commands, and managing data security [5]. The initial release of Kinetic drive has a 4TB storage capacity with 2 Ethernet connection, each of 1Gb/s. The drives can be accessed directly through an IP address with the Ethernet cable. The key and value sizes supported by the drive are up to 4KB and 1MB, respectively.

Kinetic drives communicate data in terms of key-value pairs using the Kinetic Protocol (KP) which is a network protocol used to transfer data from one host to another over a TCP-based network, such as the Internet. Host applications communicate with the Kinetic drives by sending Kinetic Protocol Data Units (Kinetic PDU) over a network using TCP. A Kinetic PDU is composed of five fields: PDU version, kinetic message length, value length, kinetic message and value. The Kinetic PDU packet is shown in Figure 2.2 and the description of each field is shown in Table 2.1

Starting Offset	Type	Length	Description
0	Byte	1 Byte	PDU Version: currently hex 46, denoting the beginning of the message.
1	4-byte big Endian Integer	4 Bytes	The number of bytes in the Kinetic Message field. The maximum length for a kinetic message is 1 MiB (2ˆ20 bytes)
5	4-byte big Endian Integer	4 Bytes	The number of bytes in the value field. The maximum length for a value is 1 MiB (2ˆ20 bytes)
9	Bytes	<1 MiB	The Kinetic Message.
9 + length of kinetic Message	Bytes	<1 MiB	The value

Table 2.1: Kinetic Protocol Data Unit Structure [4]

Function Name	Function Action
put(key,value)	stores key-value pair on the drive.
get(key)	returns the value associated to the given key.
delete(key)	deletes the entry that is associated with the key specified.
getNext(key)	gets the entry associated with a key after the given key.
getPrevious(key)	gets the entry associated with a key before the given key.
getKeyRange(start,end)	gets a list of keys in the given key range.
getMetadata(key)	gets entry metadata for the specified key.

Table 2.2: Kinetic Drive API Provided Functions

The Kinetic Protocol also defines a range of operations that can be seen in the Kinetic API. This API provides some methods to deal with key-value pair data as shown on Table 2.2 [5].

Chapter 3

TurboKV: Scaling Up The Performance of Distributed Key-Value Stores With In-Switch Coordination [1]

3.1 Introduction

Programmable switches in software-defined network promise flexibility and high throughput [18,19]. Recently, the Programming Protocol-Independent Packet Processor (P4) [14] unleashes capabilities that give the freedom to create intelligent network nodes performing various functions. Thus, applications can boost their performance by offloading part of their computational tasks to these programmable switches to be executed in the network. Nowadays, programmable networks get a bigger foot in the data center doors. Google cloud started to use the programmable switches with P4Runtime [15] to build and control their smart networks [16]. Some Internet Service Providers (ISPs), such as AT&T, have already integrated programmable switches in their networks [17]. These switches can be controlled by network operators to adapt to the current network state and application requirements.

Recently, there has been an uptake in leveraging programmable switches to improve distributed systems, e.g., NetPaxos [20, 21], NetCache [22], NetChain [23], DistCashe [24], Pegasus [35], Concordia [36] and iSwitch [25]. This uptake is due to the massive evolution on the capailities of these network switches, e.g., Tofino ASIC from Intel [26] which provides sub-microsecond per-packet processing delay with bandwidth up to 6.5 Tbs and throughput of few billions of packets processed per second and Tofino2 [27] with bandwidth up to 12.8 Tbs. These systems use the switches to provide orders of magnitude higher throughput than the traditional server-based solutions.

On the other hand, through the massive use of mobile devices, data clouds, and the rise of Internet of Things [6], enormous amount of data has been generated and analyzed for the benefit of society at a large scale. This data can be text, image, audio, video, etc., and is generated by different sources with un-unified structures. Hence, this data is often maintained in key-value storage, which is widely used due to its efficiency in handling data in key-value format, and flexibility to scale out without significant database redesign. Examples of popular key-value stores include Amazon's Dynamo [8], Redis [9], RAMCloud [10], LevelDB [11] and RocksDB [12].

Such huge amount of data can not be stored in a single storage server. Thus, this data has to be partitioned across different storage instances inside the data center. The data paritions and their storage node mapping (directory information) are either stored on a single coordinator node, e.g., the master coordinator in distributed Google file system [28], or replicated over all storage instances [8, 9, 29]. In the first approach, the master coordinator represents a single point of failure, and introduces a bottleneck in the path between clients and storage nodes; as all queries are directed to it to know the data location. Moreover, the query response time increases, and hence, the storage system performance decreases. In the second approach, where the directory information is replicated on all storage instances, there are two strategies that a client can use to select a node where the request will be sent to: *server-driven coordination* and *client-driven coordination*.

In server-driven coordination, the client routes its request through a generic load balancer that will select a node based on load information. The selected node acts like the coordinator for the client request. It answers the query if it has the data partition or forwards the query to the right instance where the data partition resides. In this

strategy, the client neither has knowledge about the storage nodes nor needs to link any code specific to the key-value storage it contacts. Unfortunately, this strategy has a higher latency because it introduces additional forwarding step when the request coordinator is different from the node holding the target data.

In client-driven coordination, the client uses a partition-aware client library that routes requests directly to the appropriate storage node that holds the data. This approach achieves a lower latency compared to the server-driven coordination as shown in [8]. In [8], the client-driven coordination approach reduces the latencies by more than 50% for both 99.9^{th} percentile and average cases. This latency improvement is because the client-driven coordination eliminates the overhead of the load balancer and skips a potential forwarding step introduced in the server-driven coordination when a request is assigned to a random node. However, it introduces additional load on the client to periodically pickup a random node from the key-value store cluster to download the updated directory information to perform the coordination locally on its side. It also requires the client to equip its application with some code specific to the key-value store used.

Another challenge in maintaining distributed key-value stores is handling dynamic workloads and coping with changes in data popularity [37, 38]. Frequent requests to hot data over cold data lead to load imbalance among storage nodes; some nodes are heavily congested while others become under-utilized. This results in a performance degradation of the whole system and a high tail latency. The most notable research in this area using the programmable switches includes NetCache [22] and Pegasus [35]. NetCache tackles the load balancing for high skewed read only workload via caching the most popular $O(nlogn)$ key-value pairs in the ToR switch. Pegasus solves the problem of load balancing through the selective replication approach. It maintains a coherent directory in the ToR switch for the most popular $O(nlogn)$ key-value pairs, and distributes the load between the storage nodes that have these replicated items. Although both of NetCache and Pegasus use the programmable switches to handle load balancing in distributed key-value stores, none of them handle the partition management problem in this environment. Also, they can only handle point queries and can not handle queries that ask for a specific key-range scan.

In this chapter, we propose *TurboKV* : a novel Distributed Key-Value Store Architecture that leverages the power and flexibility of the new generation of programmable switches to overcome the limitations of existing systems. *TurboKV* scales up the performance of the distributed key-value storage by offloading the partitions management and query routing to be carried out in network switches. *TurboKV* uses a *switch-driven coordination* which utilizes the programmable switches as: 1) partition management nodes to store and manage the directory information of key-value store; and 2) monitoring stations to measure the load of storage nodes, where this monitoring information is used to balance the load among storage nodes.

TurboKV adapts a hierarchical indexing scheme to distribute the directory information records inside the data plane of the data center network switches. It uses a key-based routing protocol to map the requested key in the query packet from the client to its target storage node by injecting some information about the requested data in packet headers. The programmable switches use this information to decide where to send the packet to reach the target storage node directly. This in-switch coordination approach removes the load of routing the requests from the client in the client-driven coordination without introducing an additional forwarding step introduced by the coordination node in the server-driven coordination. Unlike existing systems [22, 35], *TurboKV* can coordinate the requests for all query types: point queries and range scans. It also handles two different partitioning techniques: hash partitioning and range partitioning. Applications can choose any of these two partitioning techniques according to their needs.

To achieve both reliability and high availability, *TurboKV* replicates key-value pair partitions on different storage nodes. For each data partition, *TurboKV* maintains a list of nodes that are responsible for storing the data of this partition. *TurboKV* uses the chain replication [39] model to guarantee strong data consistency between all partition replicas. In case of having a failing node, requests will be served with other available nodes in the partition replica list.

TurboKV also handles load balancing by adapting a dynamic allocation scheme that utilizes the architecture of software-defined network [13, 40]. In our architecture, a logically centralized controller, which has a global view of the whole system [41], makes decisions to migrate/replicate some of the popular data items to other under-utilized

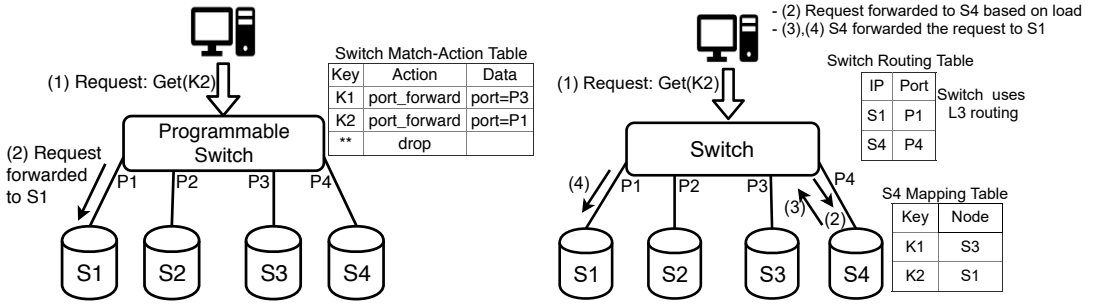
storage nodes using monitoring reports from the programmable switches. Then, it updates the switches' data plane with the new indexing records. Overall, our contributions in this chapter of the thesis are four-fold:

- We propose the in-switch coordination paradigm, and design an indexing scheme to manage the directory information records inside the programmable switch along with protocols and algorithms to ensure the strong consistency of data among replica, and achieve the reliability and availability in case of having nodes failure.
- We introduce a data migration mechanism to provide load balancing between the storage nodes based on the query statistics collected from the network switches.
- We propose a hierarchical indexing scheme based on our proposed rack scale switch coordinator design to scale up *TurboKV* to multiple racks inside the existing data center network architecture.
- We implemented a prototype of *TurboKV* using P4 on top of the simple software switch architecture BMV2 [42]. Our experimental results show that our proposed architecture improves the throughput and reduces the latency for all query types in the distributed key-value stores.

The remaining sections of this chapter are organized as follows. The Motivation is discussed in Section 3.2. Section 3.3 provides an overview of the *TurboKV* architecture, while the detailed design of *TurboKV* is presented in Section 3.4 and Section 3.5. Section 3.6 discusses how to scale up *TurboKV* inside the data center networks. *TurboKV* implementation is discussed in Section 3.7, while Section 3.8 gives an experimental evidence and analysis of *TurboKV*. Section 3.9 provides a short survey about the related work to us, and finally, *TurboKV* is concluded in Section 3.10.

3.2 Why In-Switch Coordination?

We utilize the programmable switches to scale up the performance of distributed key-value stores with the in-switch coordination because of three reasons. First, programmable switches have become the backbone technology used in modern data centers



(a) In-Switch (2 hops) (b) Server-based (4 hops)

Figure 3.1: Request Coordination Models

or rack-scale clusters that allow developers to define their own functions for packet processing and provide flexibility to program the hardware. For example, Google uses the programmable switches to build and control their data centers networks [16].

Second, request latency is one of the most important factors that affect the performance of all key-value stores. This latency is introduced through the multiple hops that the request traverses before arriving to its final destination, in addition to the processing time to fetch the desired key-value pair(s) from that destination. When the key-value store is distributed among several storage nodes, the request latency increases because of the latency introduced to locate the desired key-value pair(s) before fetching them from that location. This process is referred to as *partition management and request coordination*, which can be organized by the storage nodes (server-driven coordination) or by the client (client-driven coordination).

Because client requests already pass through network switches to arrive at their target, offloading the partition management and query routing to be carried out in the network switches with the in-switch coordination approach will reduce the latency introduced by the server-based coordination approach; the number of hops that the request will travel from the client to the target storage node will be reduced as shown in Figure 3.1. In-switch coordination also removes the load from the client in the client-based coordination approach by making the programmable switch manage all the information for the request routing.

Third, the partition mangement and request coordination are communication-bounded rather than being computation-bounded. So, the massive evolution in the capailities of

the network switches, which provide orders of magnitude higher throughput than the highly optimized servers, makes them the best option to be used as partition mangement and request cordination nodes. For example, Tofino ASIC from Intel [26] provides few billions of packets processed per second with 6.5 Tbps bandwidth. Such performance is orders of magnitude higher than NetBricks [43] which processes millions of packets per second and has 10-100 Gbps bandwidth [23].

3.3 *TurboKV* Architecture Overview

TurboKV is a new architecture of the future distributed key-value stores that leverages the capability of programmable switches. *TurboKV* uses an in-switch coordination approach to maintain the partition management information (directory information) of distributed key-value stores, and route clients' search queries based on their requested keys to the target storage nodes. Figure 3.2 shows the architecture of *TurboKV* which consists of the programmable switches, controller, storage nodes, and system clients.

Programmable Switches. Programmable switches are the essential component in our new proposed architecture. We augment the programmable switches with a key-based routing approach to deliver *TurboKV* query packets to the target key-value storage node. We leverage match-action tables and switch's registers to design the in-switch coordination where the partition management information will be stored on the path from the client to the storage node. This directory information represents the routing information to reach one of the storage nodes (Section 3.4.1). Following this approach, the programmable switches act as request coordinator nodes that manage the data partitions and route requests to target storage nodes.

In addition to using the key-based routing module, other packets are processed and routed using the standard L2/L3 protocols which makes *TurboKV* compatible with other network functions and protocols (Section 3.4.2). Each programmable switch has a query statistics module to collect information about each partition's popularity to estimate the load of storage nodes (Section 3.5.1). This is vital to make data migration decisions to balance the load among storage nodes especially to handle dynamic workloads where the key-value pair popularity changes overtime.

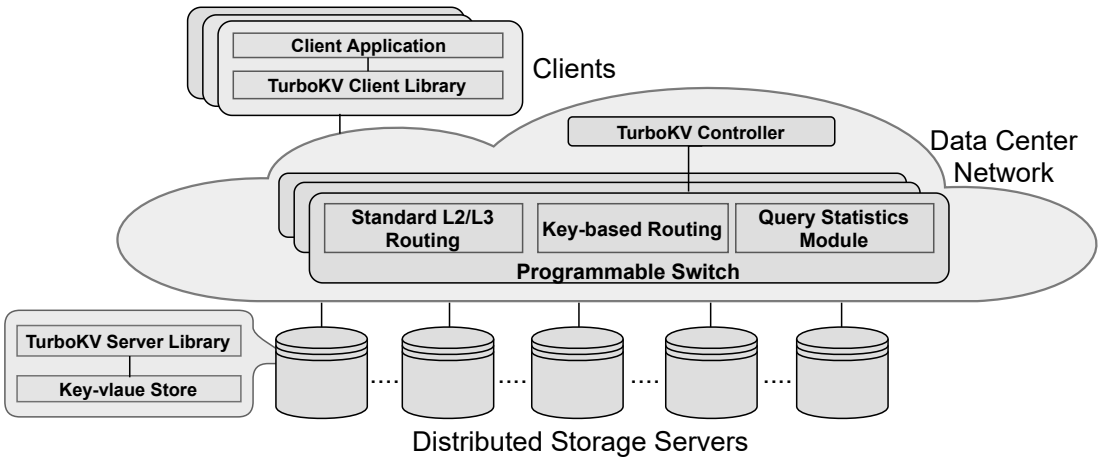


Figure 3.2: *TurboKV* Architecture

Controller. The controller is primarily responsible for system reconfigurations including (a) achieving load balancing between the distributed storage nodes (Section 3.5.1), (b) handling failures in the storage network (Section 3.5.2), and (c) updating each switch's match-action tables with the new location of data. The controller receives periodic reports form switches about the popularity of each data partition. Based on these reports, it decides to migrate/replicate part of the popular data to another storage node to achieve load balancing. Through the control plane, the controller updates the match-action tables in the switches with the new data locations. *TurboKV* controller is an application controller that is different from the network controller in SDN, and it does not interfere with other network protocols or functions managed by the SDN controller. Our controller only manages the key-range based routing and data migrations and failures associated with them. Both controllers can be co-located on the same server, or on different servers.

Storage Nodes. They represent the location where the key-value pairs reside in the system. The key-value pairs are partitioned among these storage nodes (Section 3.4.1). Each storage node runs a simple shim that is responsible for reforming *TurboKV* query packets to API calls for the key-value store, and handling *TurboKV* controller's data migration requests between the storage nodes. This layer makes it easy to integrate our design with existing key-value stores without any modifications to the storage layer.

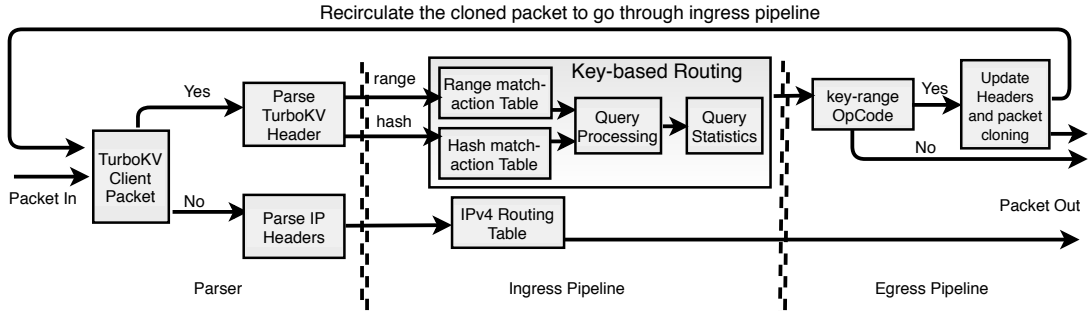


Figure 3.3: Logical View of *TurboKV* Data Plane Pipeline

System Clients. *TurboKV* provides a client library which can be integrated with the client applications to send *TurboKV* packets through the network, and access the key-value store without any modifications to the application. Like other key-value stores such as LevelDB [11] and RocksDB [12], the library provides an interface for all key-value pair operations that is responsible for constructing the *TurboKV* packets and translates the reply back to the application.

3.4 *TurboKV* Data Plane Design

The data plane provides in-switch coordination model for the key-value stores. In this model, all partition management information and query routing are managed by the switches. Figure 3.3 represents the whole pipeline that the packet traverses inside the switch before being forwarded to the right storage node. In this section, we discuss how the switch data plane supports these functions.

3.4.1 On-Switch Partition Management

Data Partitioning

In large-scale key-value stores, data is partitioned among storage nodes. *TurboKV* supports two different partitioning techniques. Applications can choose any of these two partitioning techniques according to their needs.

Range Partitioning. In range partitioning, the whole key space is partitioned into small disjoint sub-ranges. Each sub-range is assigned to a node (or multiple nodes

depending on the replication factor). The data with keys fall into a certain sub-range is stored on the same storage node that is responsible for this sub-range. The key itself is used to map the user request to one of the storage nodes. Each storage node can be responsible for one or more sub-ranges. Each storage node has LevelDB [11] or any of its enhanced versions installed where keys are stored in lexicographic order on SSTs (Sorted String Tables). This key-value library is responsible for managing the sub-ranges associated to the storage node and handling the key-value pair operations directed to the storage node. The advantage of this partitioning scheme is that range queries on keys can be supported, but unfortunately, this scheme suffers from load imbalance problem discussed in Section 3.5.1. Some sub-ranges contain popular keys which receive more requests than others. The mapping table for this type of partitioning represents the key range and the associated nodes where the data resides as shown in Figure 3.4(a). Support of range partitioning in the switch data plane is discussed in Section 3.4.1.

Hash Partitioning. In hash partitioning, each key is hashed into a 20-byte fixed-length digest using RIPEMD160 [44] which is an extremely random hash function, ensures that records are distributed uniformly across the entire set of possible hash values. We developed a variation of the consistent hashing [45] to distribute the data over multiple storage nodes. The whole output range of the hash function is treated as a fixed space. This space is partitioned into sub-ranges, each sub-range represents a consecutive set of hashing values. These sub-ranges are distributed evenly on the storage nodes. Each storage node can be responsible for one or more sub-ranges based on its load, and each sub-range is assigned to one node (or multiple nodes depending on the replication factor). Like the consistent hashing, partitioning one of the sub-ranges or merging two sub-ranges due to the addition or removal of nodes affects only the nodes that hold these sub-ranges and other nodes remain unaffected. Each data item identified by a key is assigned to a storage node if the hashed value of its key falls in the sub-range of hashing values which the storage node is responsible for. This method requires a mapping table, shown in Figure 3.4(b), where each record represents a hashed sub-range and its associated nodes where data resides. Support of hash partitioning in the switch data plane is discussed in Section 3.4.1. The disadvantage of this technique, like all hashing partitioning techniques, is that range queries can not be supported. On

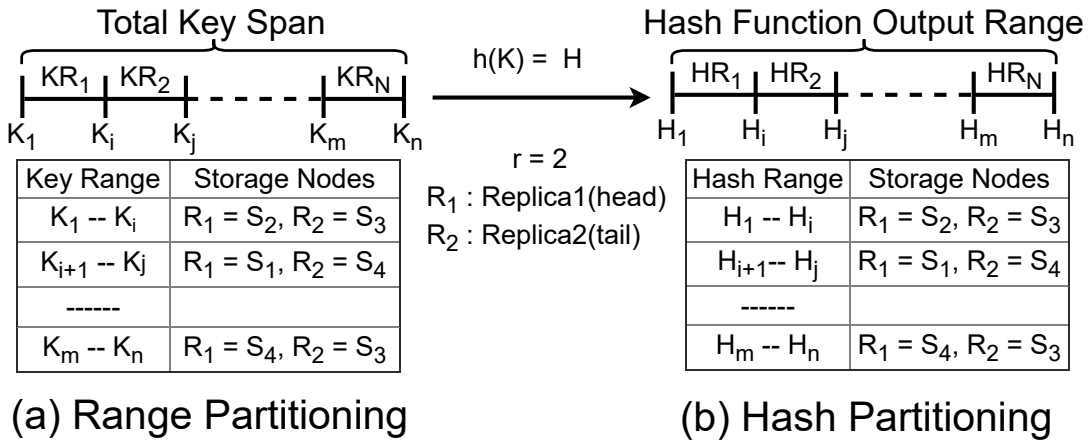


Figure 3.4: *TurboKV* Data Partitioning and Replication

the storage nodes side, data is managed in hash-tables and collisions are handled using separate chaining in the form of binary search tree.

For both partitioning techniques, we assume that there is no fixed space assigned to each sub-range (partition) on the storage node (i.e., the space that each partition consumes on a storage node can grow as long as there is available space on the storage node). Unfortunately, multiple insertions to the same partition may exceed the capacity of the storage node. In this case, The sub-range of this partition will be divided into two smaller sub-ranges. One of these small sub-ranges will be migrated to another storage node with available space. Other storage nodes that have the same divided sub-range with available space keep the data of the whole sub-range and manipulate it as it is. The mapping table will be updated with the new changes of the divided sub-ranges.

Data Replication

To achieve high availability and durability, data is replicated on multiple storage nodes. The data of each partition (sub-range) is replicated as one unit on different storage nodes. The replication factor (r) can be adjusted according to application needs. The list of nodes that is responsible for storing a particular partition is called the *replica list*. This *replica list* is stored in the mapping table as shown in Figure 3.4.

TurboKV follows the chain replication (CR) model [39]. Chain replication [39] is a form of primary backup protocols to control clusters of fail-stop storage servers.

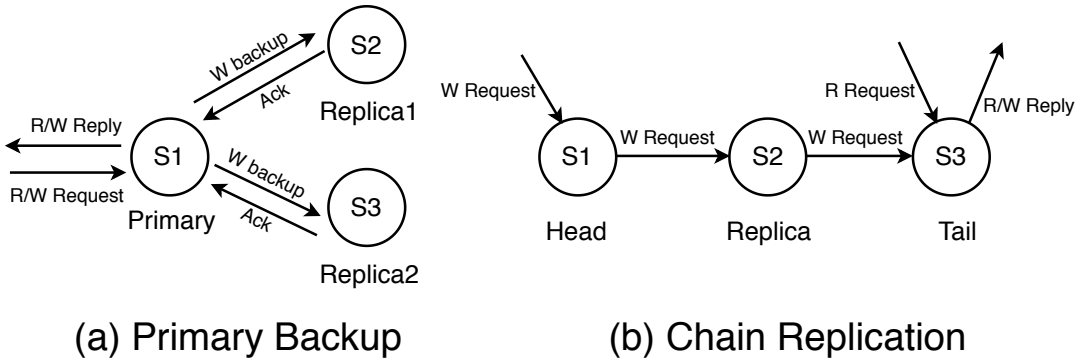


Figure 3.5: Replication Models

This approach is intended for supporting large-scale storage services that exhibit high throughput and availability without sacrificing strong consistency guarantees. In this model as shown in Figure 3.5(b), the storage nodes, holding replicas of data, are organized in a sequential chain structure. Read queries are handled by the tail of the chain, while write queries are sent to the head of the chain, the head process the request and forwarded it to its successor in the chain structure. This process continues till reaching the tail of the chain. Finally, the tail processes the request and replies back to the client.

In CR, Each node in the chain needs to know only about its successor, where it forwards the request. This makes the CR simpler than the classical primary backup protocol [46], shown in Figure 3.5(a) which requires the primary node to know about all replicas and keep track of all acknowledgement received from all replicas. Moreover, In chain replication, write queries also use fewer messages than the classical primary backup protocol, (n+1) instead of (2n) where n is number of nodes. With r replicas, *TurboKV* can sustain up to (r-1) node failures as requests will be served with other replicas on the chain structure.

Management Support in Switch Data Plane

In *TurboKV*, the switch data plane has three types of match-action tables: range partition management table, hash partition management table and the normal IPv4 routing table. We will discuss the design of the partition management tables which are related to the key-based routing protocol for the rack scale cluster shown in Figure 3.6(a). The design of both tables is the same but the value we used for matching and the value

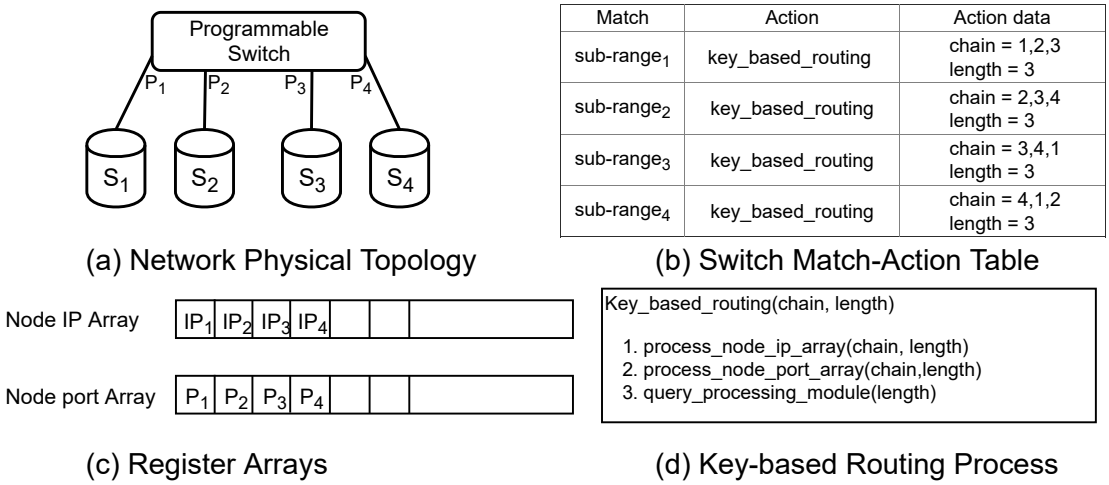


Figure 3.6: *TurboKV* Partition Management inside Switch

we match against are different based on the type of partitioning. We will refer to the value we use for matching as the *matching value*, this value represents the key itself in case of range partitioning and the hashed value of the key in case of hash partitioning. The partition management match-action table design is shown in Figure 3.6(b). Each record in the table consists of three parts: match, action and action data. The match represents the value that we match the *matching value* against, we refer to it as a *sub-range*. This *sub-range* represents the start and end keys of a sub-range from the whole key span in range partitioning, or the start and end hash values of a consecutive set of hash values in hash partitioning. The action represents the key-based routing that will be executed when a *matching value* falls within the *sub-range*. The action data consists of two parts: chain and length. Chain represents the forwarding information for nodes forming the chain of the sub-range. This information includes node's IP address and the port from the switch to the storage node. Nodes' information is sorted according to node's position in the chain structure (i.e., first node is the head and last node is the tail), and is used in updating packet during the action execution.

In *TurboKV*, each node holds the data of one or more sub-ranges. This makes each node's forwarding information appears more than once in the match-action table records. *TurboKV* uses two arrays of registers in the switch's data plane to save the forwarding information: node IP array, and node port array. For each storage node,

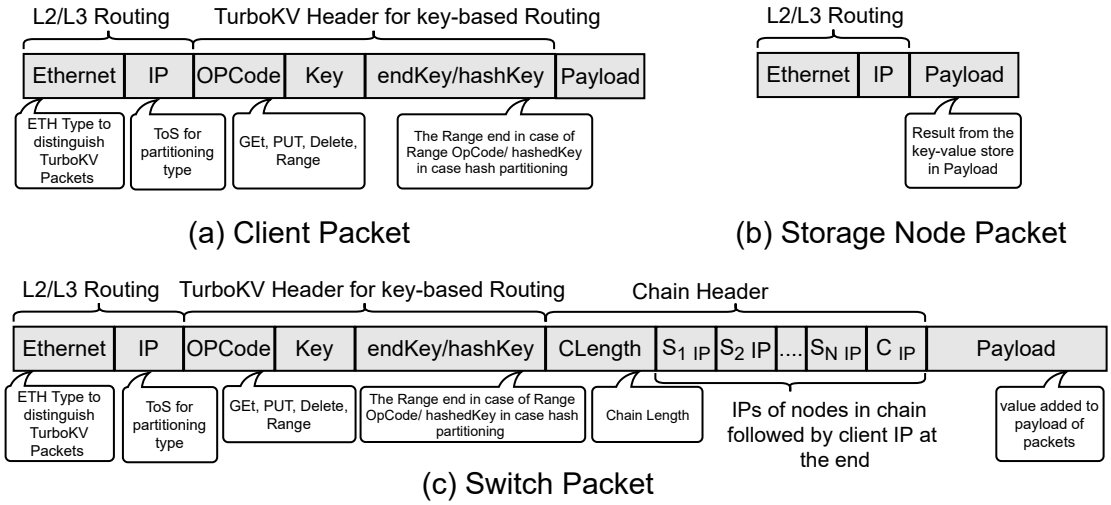


Figure 3.7: *TurboKV* Packet Format

the forwarding information is stored at the same index in the two arrays as shown in Figure 3.6(c). For example, the information of storage node *S1* is stored at index 1 in the two arrays and the same for the other storage nodes. The index of the storage nodes in the register arrays is stored as action data in the match-action table records to form the chain as shown in Figure 3.6(b). The key-based routing uses these indexes to process the register arrays and fetch the forwarding information for the replica nodes. This information is used by the query processing module to forward packets to their next hop.

3.4.2 Network Protocol Design

Packet Format. Figure 3.7(a) shows the format of *TurboKV* request packet sent from clients. The programmable switches use the `Ethernet Type` in the Ethernet header to identify *TurboKV* packets, and execute the key-based routing. Other switches in the network do not need to understand the format of *TurboKV* header, and treat all packets as normal IP packets. The `ToS` (Type of Service) in the IP header is used to distinguish between three types of *TurboKV* packets: range partitioned data packet, hash partitioned data packet, and *TurboKV* packet previously processed by the switch. The *TurboKV* header consists of three main fields: `OpCode`, `Key`, and `endKey/hashedKey`. The `OpCode` is a one-byte field which stands for the code of the key-value operation

(*Get, Put, Del, and Range*). `Key` stores the key of a key-value pair. In *Range* operation, `Key` and `endKey/hashedKey` are used to represent the start and end of the key range, respectively. In case of hash partitioning, the `endKey/hashedKey` is set with the hashed value of the key to perform the routing based on it instead of the key itself.

After the client packet is processed by the programmable switch, the switch adds the chain header shown in Figure 3.7(c). This header is used by the storage nodes for the chain replication model. It includes two fields. The first field is the number of nodes which the packet passes by in the chain including the client IP (`CLength`). The second field has these nodes' IP addresses ordered according to their position in the chain followed by the client IP at the end. The packet format of the reply from storage node to client is a standard IP packet shown in Figure 3.7(b). The result is added to the packet payload.

Network Routing. *TurboKV* uses a key-based routing protocol to route packets from clients to storage nodes. The client sends the packet to the network. Once it reaches the programmable switch, the switch extracts the `Key` (in case of range partitioning) or the `endkey/hashedKey` (in case of hash partitioning) from *TurboKV* header, and looks up the corresponding key-based match-action table using the value of the extracted field (*matching value*). If there is a hit, the switch fetches the chain nodes' information from the registers based on the specified indexes in the match-action table. Then, the switch processes this information based on the query type (`OpCode`). After that, the switch updates the packet with the target node's information, and forwards it to the next hop on the path to this target node. The switch uses the *range matching* for table lookup, in which, it matches the *matching value* against the sub-range in the corresponding key-based match-action table. If the *matching value* falls within one of the sub-ranges (hit), the key-based routing action is processed using the action data. The programmable switches route the previously processed *TurboKV* packets and the storage node to client packets using the standard L2/L3 protocol without passing the key-based routing and perform the match-action based on the destination IP in the IP header.

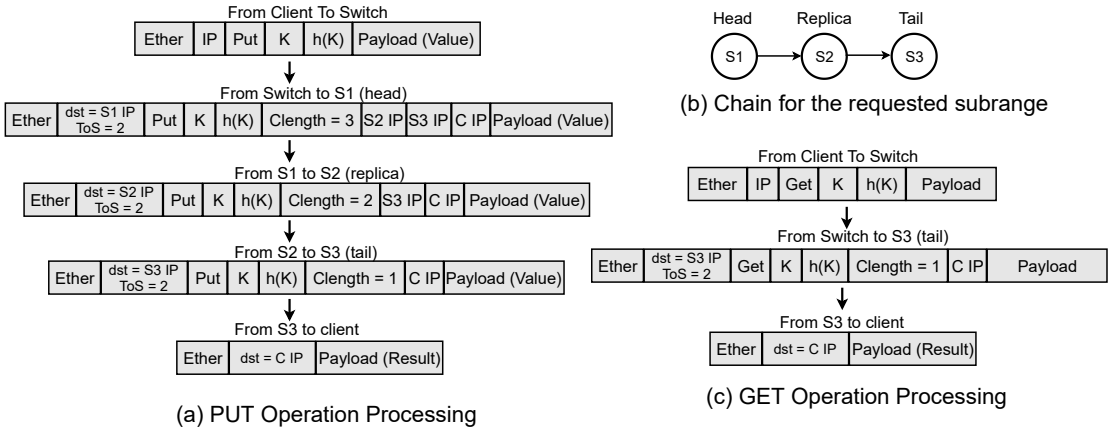


Figure 3.8: *TurboKV* KV Storage Operations

3.4.3 Key-value Storage Operations

PUT and DELETE Queries. In chain replication, PUT and DELETE queries are processed by each node along the chain starting from the head and replied by the tail. On the switch side, after processing one of the key-based match-action tables with the *matching value* and fetching the corresponding chain information from the register arrays, the query processing module sets the destination IP address in the IP header with the IP address of the chain head node and changes the `ToS` value to mark the packet as previously processed. The egress port is set with the forwarding port of the chain head node, then the chain header is added to the packet with `CLength` equals to the length of the chain and the chain nodes' IP addresses ordered according to their position in the chain followed by the client IP at the end as shown in Figure 3.8(a). Finally, the packet is forwarded to the head of chain node. As shown in Figure 3.8(a), when the packet arrives at a storage node, it is processed by *TurboKV* storage library. The node updates its local copy of the data. Then, it reads the chain header, sets the destination address in the IP header with the IP of its successor and reduces the `CLength` by 1, then forwards the packet to its successor. Packets received by the tail node, with `CLength` = 1, have their chain header and *TurboKV* header removed, and the result is sent back using the client IP address as the destination address in the IP header.

Algorithm 1 Range Query Handling

```
1: Input pkt: packet entering the egress pipeline
2: matched_subrange: the subrange where the start key of the requested range falls

3: Output pkt_out: packet to be forwarded to the next hop
4: Output pkt_cir: packet to be circulated and sent to ingress pipeline as new packet

5: Begin:
6: pkt_out = pkt      // clone the packet
7: if pkt.OpCode == range then
8:    // check if range spans multiple nodes
9:    if pkt.request.endKey ¿ matched_subrange.endKey then
10:      pkt_cir = pkt      // clone the packet
11:      pkt_out.request.endKey = matched_subrange.endKey
12:      pkt_cir.request.Key = Next(matched_subrange.endKey)
13:   end if
14: end if
15: if pkt_cir.exist() then
16:   circulate(pkt_cir)      // send packet to ingress pipeline again
17: end if
18: send_to_output_port(pkt_out)
```

GET Queries. Following the chain replication, GET queries are handled by the tail of the chain. After performing the matching on the *matching value* and fetching the corresponding chain information from the register arrays, the query processing module sets the destination IP address in the IP header with the IP address of the chain tail node and changes the `ToS` value to mark the packet as previously processed. The egress port is set with the forwarding port of the chain tail node, then the chain header is added to the packet with `CLength` equals to 1 and one node IP which represents the client IP as shown in Figure 3.8(c). Finally, the packet is forwarded to the storage node. When the packet arrives at the storage node, it is processed by *TurboKV* storage library. The query result is added to the payload of the packet. and the client IP is popped up from the chain header and put in the destination address in the IP header. The chain and *TurboKV* headers are removed from the packet and the result is sent back to the client.

Range Queries. If the data is range partitioned, *TurboKV* can handle range queries. The requested range in the *TurboKV* header may span multiple storage nodes. Thus, the switch divides the range into several sub-ranges, each sub-range corresponds to a packet. Each of these packets contains the start and end keys of the corresponding sub-range. Each packet is handled by the switch like a separate read query and forwarded to the tail node of its partition's chain. Unfortunately, the switch cannot construct new packets from scratch on its own. Therefore, in order to achieve the previous scenario, we placed the range operation check in the egress pipeline as shown in Figure 3.3. We use the `clone` and `circulate` operations, supported in the architecture of the programmable switches and P4, to solve the packet construction problem. When a range operation is detected in the egress pipeline, the packet is processed as shown in Algorithm 1.

3.5 *TurboKV* Control Plane Design

The control plane provides load balancing module based on the query statistics collected in the data plane. It also provide a failure handling module to guarantee availability and fault tolerance. In this section, we discuss how the control plane supports these functions.

3.5.1 Query Statistics and Load Balancing

In *TurboKV*, the data plane has a query statistics module to provide query statistics reports to the *TurboKV* controller. Thus, the controller can estimate the load of each storage node, and make decisions to migrate part of the popular data to one of the under-utilized storage nodes. As shown in Figure 3.9(a), the switch's data plane maintains a per-key range counter for each key range in the match-action table. Upon each hit for a key range, its corresponding counter is incremented by one. The controller receives reports periodically from the data plane including these statistics, and resets these counters in the beginning of each time period. Then, the controller compares the received statistics with the specifications of the storage nodes. If a storage node is over-utilized (the number of requests directed to the storage node is greater than its available bandwidth which results in high tail latency), the controller migrates a subset of the hot data in a sub-range to another under-utilized storage node with available bandwidth and

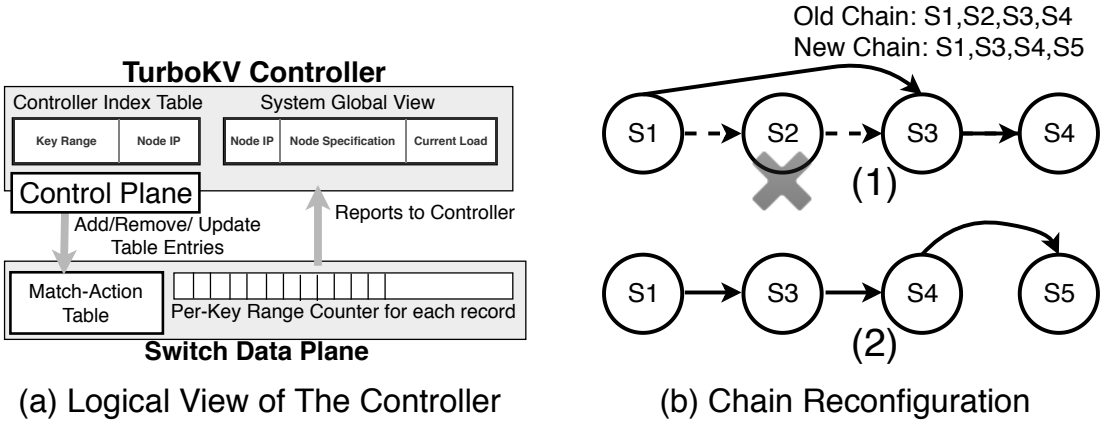


Figure 3.9: *TurboKV* Control Plane

reconfigures the chain of the updated sub-range. Then, the controller updates records in the match-action table of the switches with the new chain configurations. After the sub-range's data is migrated to other storage nodes, the old copy is removed from the over-utilized one. Currently, the controller follows a greedy selection algorithm to select the least utilized node where data will be migrated.

TurboKV uses the physical migration of the data to achieve load balancing between the storage nodes. This approach adapts with all kinds of workloads compared to the caching approach. In caching [22], the cache absorbs the read requests of the very popular key-value pairs, which makes it performs well in highly skewed read-only workload, but the effect of caching in load balancing decreases if the workload's ratio of updates increases. This behavior resulted from the invalid cached key-value pairs, which make the request directed to the target storage node to retrieve the valid pairs.

3.5.2 Failures Handling

We assume that the controller process is a reliable process and it is different from the SDN controller. We also assume that links between storage nodes and switches in data centers are reliable and are not prone to failures.

Storage Node Failure. When the controller detects a storage node failure, it reconfigures the chains of the sub-ranges on the failed storage node and updates their corresponding records in the key-based match-action table through the control plane.

The controller removes the failed storage node from its position in all chains. In each chain, the predecessor of the failed node will be followed by the successor of the failed node, reducing the chain length by 1 as shown in Figure 3.9(b). If the failed node was the head of the chain, then the new head will be its successor. If the failed node was the tail of the chain, then the new tail will be its predecessor. Reducing the chain length by 1 makes the system able to sustain less number of failures. That is why the controller distributes the data of the failed node in sub-range units among other functional nodes, and adds these new nodes at the end of these sub-ranges' chains in their corresponding records in the key-based match-action table. This process of sub-range redistribution restores the chain to its original length.

Switch failure. The storage servers in the rack of the failed switch would lose access to the network. The controller will detect that these storage servers are unreachable. The controller treats these storage servers as failed storage nodes, and distributes the load of these storage servers among other reachable servers as described before. Then, the failed switch needs to be rebooted or replaced. The new switch starts with an index table which contains all the key ranges handled by its connected storage servers.

3.6 Scaling Up to Multiple Racks

We have discussed the in-switch coordination for distributed key-value stores within a rack of storage nodes with the Top-of-Rack (ToR) switch as the coordinator. We now discuss how to scale out the in-switch coordination in the data center network. Figure 3.10 shows the network architecture of data centers. All the servers in the same rack are connected by a ToR switch. In the highest level, there are Aggregate switches (AGG) and Core switches (Core).

To scale out distributed key-value stores with in-switch coordination, we develop a "hierarchical indexing" scheme. Each ToR switch has the directory information of all sub-ranges located on its connected storage nodes as described before in Figure 3.6. In addition to the IPv4 routing table, each AGG switch has two range match-action tables (range and hash), where each table consists of the sub-ranges in its connected ToR switches. The Core switches have the range match-action tables of sub-ranges in its connected AGG switches. With each sub-range in either the AGG or Core switches,

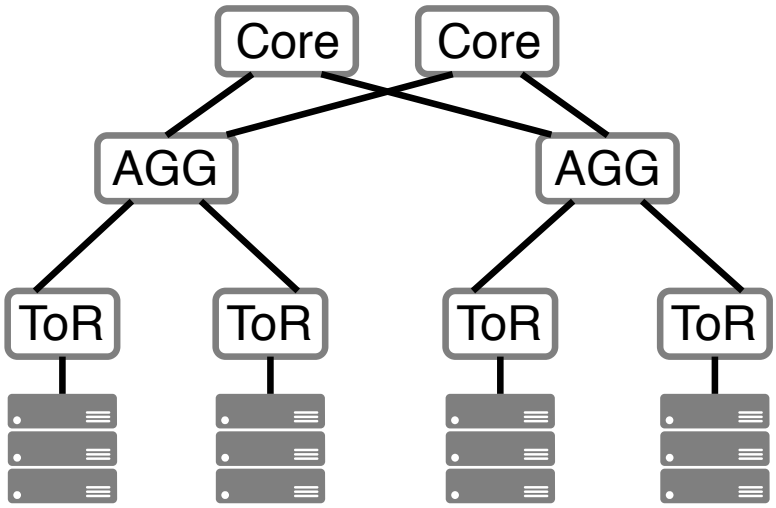


Figure 3.10: Scaling Up inside Data center Network

the action data represents only the forwarding port towards the head or the tail of the sub-range's chain. No chains are stored in these switches. When a packet is received by an AGG switch or core switch, this packet is processed by the key-based routing protocol without adding any chain header to the packet and forwarded to one of the ports towards the head or tail of the sub-range chain based on the query (write or read respectively). When the packet arrives at ToR switch, the switch processes the packet as discussed in Section 3.4.3. Replicas of a specific sub-range may be located on different racks. This design leverages the existing data center network architecture and does not introduce additional network topology changes.

3.7 Implementation Details

We have implemented a prototype of *TurboKV*, including all switch data plane and control plane features, described in Section 3.4 and Section 3.5. We have also implemented the client and server libraries that interface with applications and storage nodes, respectively, as described in Section 3.3.

Due to lack of real hardware, the switch data plane is written in P4 and is compiled to the simple software switch BMV2 [42] running on Mininet [47]. The key size of the key-value pair is 16 bytes with total key range spans from 0 to 2^{128}. This range is

divided into index records which are saved on the switch data plane. We used 4 register arrays, one for saving the storage nodes' IP addresses, one for saving the forwarding port of the storage nodes, one for counting the read access requests of the indexing records and the last one for counting the update access requests of the indexing records. The controller is able to update/read the values of these registers through the control plane. It also can add or remove table entries to balance the load of the storage nodes. The switch data plane does not store any key-value pairs as these pairs are saved on the storage nodes. This approach makes *TurboKV* consumes a small amount of the on-chip memory leaving enough space for processing other network operations. The controller is written in Python and can update the switch data plane through the switch driver by using the Thrift API generated by the P4 compiler.

The client and server libraries are written in Python using Scapy [48] for packet manipulation. The client can translate a generated YCSB [38] workload with different distributions and mixed key-value operations into *TurboKV* packets' format and send them through the network. We used Plyvel [49] which is a Python interface for levelDB [11] as the storage agent. The server library translates *TurboKV* packets into Plyvel format and connects to levelDB to perform the key-value pair operations. We used chain replication with chain length equals to 3 for data reliability.

3.8 Performance Evaluation

This section provides the experimental results of *TurboKV*. We show the performance improvement of *TurboKV* on the key-value operations latency and system throughput.

Experimental Setup. Our experiments consist of eight simple software switches BMV2 [42] connecting 16 storage nodes and 4 clients as shown in Figure 3.11. Each of the clients ($h_{17}, h_{18}, h_{19}, h_{20}$) runs the client library and generates the key-value queries. These clients represent the request aggregation servers who are the clients of the storage nodes in data centers. Each storage node (h_1, h_2,...., h_{15}, and h_{16}) runs the server library and uses LevelDB as the storage agent. The whole topology runs on Mininet. The data is distributed over the storage nodes using the range partitioning described in Section 3.4.1 with 128 records index table. Each storage node is responsible for 24 sub-ranges (head of chain of 8 sub-ranges, replica for 8 sub-ranges and tail of chain for

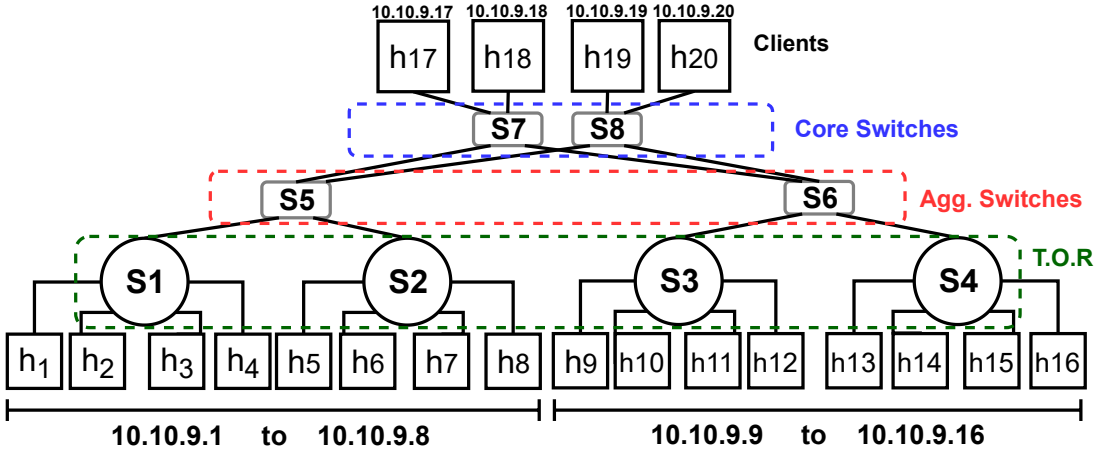


Figure 3.11: TurboKV Experiment Topology

8 sub-ranges).

Comparison. We compared our in-switch coordination (*TurboKV*) with server-driven coordination (S-Coord) and client-driven coordination (C-Coord) described in Section 3.1. In *TurboKV*, the directory information is stored in the switch data plane and updated by *TurboKV* controller, also the key-based routing is used to route the query from client to target storage node. In server-driven coordination which is implemented on most of existing key-value stores [8,29], all the storage nodes store the directory information and can act as the request coordinator. In client-driven coordination, the client acts as the request coordinator. The client has to download the directory information periodically from a random storage node, because, with lots of outdated directory information, the client-driven coordination tends to act as the server-driven coordination as the wrong storage node that the client contacts will forward the request again to the target storage node. Both of client-driven and server-driven approaches route the query using the standard L2/L3 routing protocols.

Note that in our experiments, we compare with the ideal case of the client driven coordination where the client has the updated directory information and sends the query directly to the target storage node, ignoring the latency introduced by pulling this information periodically from a random storage node because this latency will depend on the client location and also the load of the storage node that the client contacts

to pull this information. The ideal case of the client-driven coordination represents *the least latency* that the client's request can achieve because it represents the direct path from the client to the target storage node ignoring any latency resulted from having outdated directory information which may cause extra forwarding steps in the path from the client to the target storage node. With lots of updates to the directory information of the key-value store, the ideal client-driven coordination can not be achieved in real life systems.

Workloads. We use both uniform and skewed workloads to measure the performance of *TurboKV* under different workloads. The skewed workloads follow Zipf distribution with different skewness parameters (0.9, 0.95, 1.2). These workloads are generated using YCSB [38] basic database with 16 byte key size and 128 byte value size. The generated data is stored into records' files and queries' files, then parsed by the client library to convert them into *TurboKV* packet format. We generate different types of workloads: read-only workload, scan-only workload, write-only workload and mixed workload with multiple write ratios.

3.8.1 Effect on System Throughput

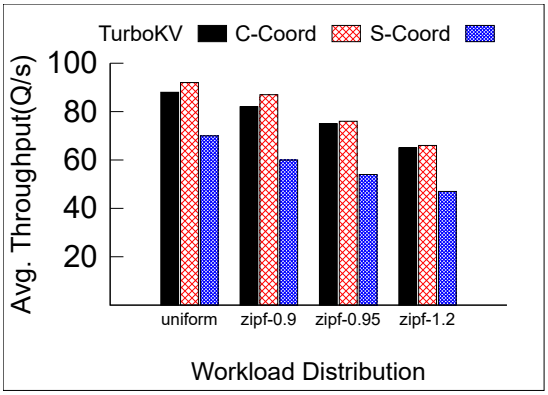
Impact of Read-only Workloads. Figure 3.12(a) shows the system throughput under different skewness with read-only queries. We compare *TurboKV* vs the server-driven coordination (S-Coord) and the ideal client-driven coordination (C-Coord). In Figure 3.12(a), *TurboKV* performs nearly the same as the ideal client-driven coordination (C-Coord) in the highly skewed workload (zipf-0.95 and zipf-1.2), and less than the ideal client-driven coordination (C-Coord) by maximum of 5% in the uniform and zipf-0.9 workloads. This result is because *TurboKV* manages all directory information in the switch data plane and uses the key-based routing to deliver the requests to the storage nodes directly. Moreover, *TurboKV* eliminates the load of downloading the updated directory information periodically from storage nodes, as this part is managed by the controller who will update the directory information in the switch data plane through the control plane. In addition, *TurboKV* outperforms the server-driven coordination (S-Coord) and improves the system throughput with minimum of 26% and maximum of 39%. This result is because *TurboKV* eliminates the overhead of the load balancer and

skips a potential forwarding step introduced in the server-driven coordination (S-Coord) when a request is assigned to a random storage node.

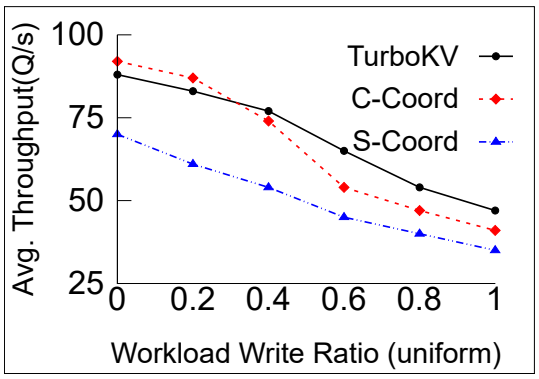
Impact of Write Ratio. Figure 3.12(b) and Figure 3.12(c) show the system throughput under uniform and skewed workload with varying the workload write ratio. As shown in Figure 3.12(b) and Figure 3.12(c), the throughput decreases as the write ratio increases for the three approaches, because each write query has to update all the copies of the key-value pair following the chain replication approach before returning the reply to the client. *TurboKV* performs roughly the same as the ideal client-driven coordination (C-Coord) in the workloads with low write ratio, but *TurboKV* outperforms the ideal client-driven coordination (C-Coord) as the write ratio increases. This behavior is because of the chain replication implemented in the system for availability and fault tolerance. In *TurboKV*, the switch inserts all the chain nodes in the packet. When the packet arrives at a storage node, the node updates its local copy and forwards the request to the next storage node directly without any further mapping to know its chain successor. But, in the ideal client-driven coordination (C-Coord), the client sends the write query to the head of chain's node. When the query arrives at the storage node, the node updates its local copy and then accesses its saved directory information to know its chain successor, then forwards the packet to it. So, when the write ratio increases, this scenario is performed for larger portion of queries which affects the system throughput. Also, *TurboKV* outperforms the server-driven coordination (S-Coord) by minimum of 26% and maximum of 44% in case of uniform workload, and by minimum of 37% and maximum of 47% in case of the skewed workload. This improvement is because of the elimination of the forwarding step when the request is assigned to a random storage node and also the elimination of further mapping steps on each storage node for knowing the chain successor.

3.8.2 Effect on Key-value Operations Latency

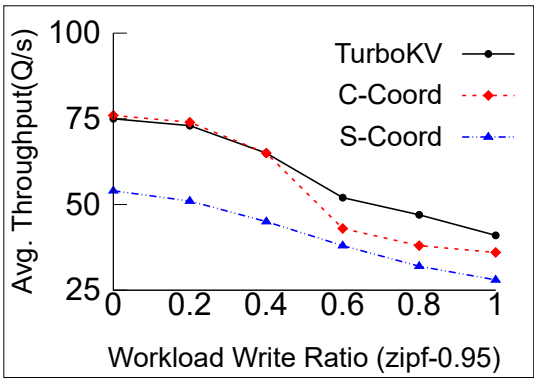
Figures 3.13, 3.14 and 3.15 show the cumulative distribution function (CDF) of all key-value operations latencies under uniform and Zipf-1.2 workloads for *TurboKV*, ideal client-driven coordination (C-Coord) and server-driven coordination (S-Coord). The



(a) Throughput vs Skewness - Read only

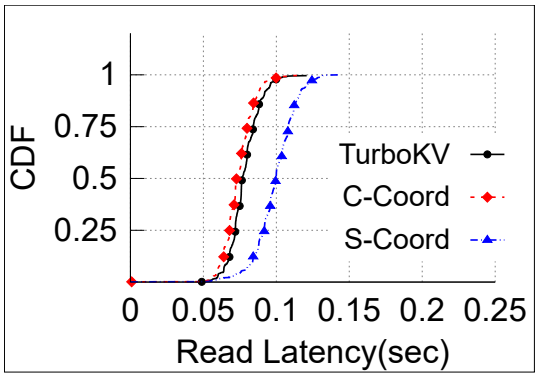


(b) Throughput vs Write Ratio - Uniform

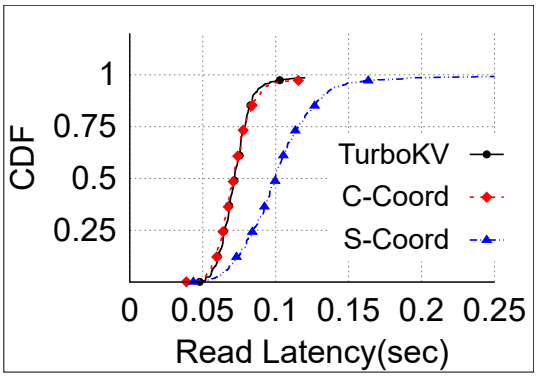


(c) Throughput vs Write Ratio - zipf0.95

Figure 3.12: TurboKV Effect on System Throughput



(a) Uniform Workload

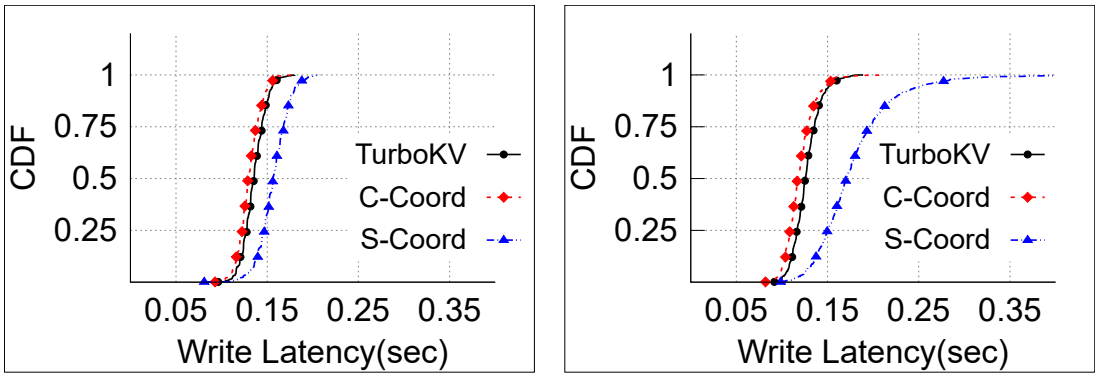


(b) Zipf-1.2 Workload

Figure 3.13: TurboKV Effect on Read Latency

	Read - Uniform Workload (Get) (msec)			Read - Zipf1.2 Workload (Get) (msec)		
	Mean	50th Percentile	99th Percentile	Mean	50th Percentile	99th Percentile
In-Switch Coordination (TurboKV)	72.5	71.2	103.3	72.2	71.8	105.3
Client-driven Coordination (C-Coord)	69.8	68.8	98.6	71.4	70.9	104
Server-driven Coordination (S-Coord)	86.6	87.7	127.8	102.8	99.8	206.8

Table 3.1: TurboKV Read Request Latency Analysis

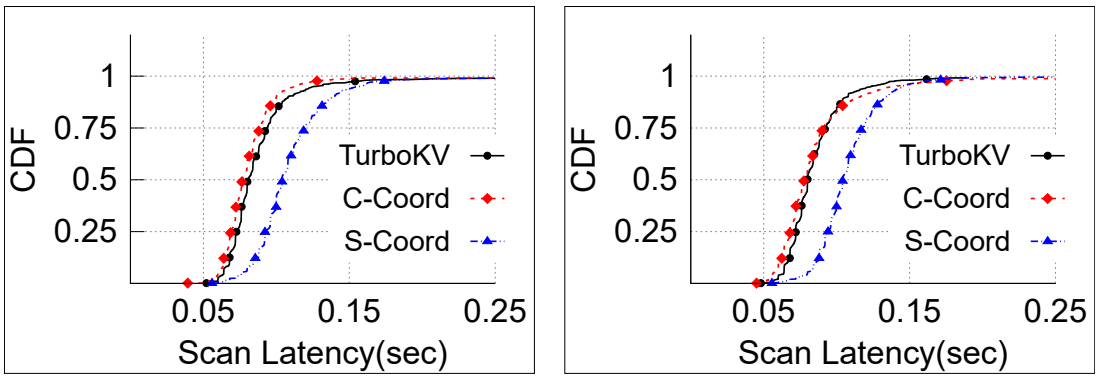


(a) Uniform Workload (b) Zipf-1.2 Workload

Figure 3.14: TurboKV Effect on Write Latency

	Write - Uniform Workload (Put) (msec)			Write - Zipf1.2 Workload (Put) (msec)		
	Mean	50th Percentile	99th Percentile	Mean	50th Percentile	99th Percentile
In-Switch Coordination (TurboKV)	123.5	121.8	165.8	126.8	125.4	172.4
Client-driven Coordination (C-Coord)	117.5	116.1	153.5	119.7	117.2	167.3
Server-driven Coordination (S-Coord)	138.2	138	189	178.3	170.9	330.6

Table 3.2: TurboKV Write Request Latency Analysis



(a) Uniform Workload (b) Zipf-1.2 Workload

Figure 3.15: TurboKV Effect on Scan Latency

	Scan - Uniform Workload (Range) (msec)			Scan - Zipf1.2 Workload (Range) (msec)		
	Mean	50th Percentile	99th Percentile	Mean	50th Percentile	99th Percentile
In-Switch Coordination (TurboKV)	84.3	80	160.2	87.3	80.3	204
Client-driven Coordination (C-Coord)	80.8	76.5	139	85.6	80	196
Server-driven Coordination (S-Coord)	109	104	184.5	112	104.5	242.6

Table 3.3: TurboKV Scan Request Latency Analysis

analysis of these three figures is shown in Table 3.1, Table 3.2 and Table 3.3, respectively. Figures 3.13(a), 3.13(b), 3.14(a), and 3.14(b) show that the read and write latencies in *TurboKV* are very close to the ideal client-driven coordination (C-Coord) for uniform and skewed workload, because *TurboKV* skips potential forwarding step like the ideal client-driven coordination (C-Coord) by managing the directory information in the switch itself. Compared to server-driven coordination (S-Coord), *TurboKV* reduces the read latency by 16.3% on the average and 19.2% for the 99^{th} percentile for the uniform workload, and by 30% on the average and 49% for the 99^{th} percentile for the skewed workload. *TurboKV* also reduces the write latency by 11% on the average and 12.3% for the 99^{th} percentile for the uniform workload, and by 29% on the average and 48% for the 99^{th} percentile for the skewed workload. The reduction in skewed workload is larger than the reduction in uniform workload, because *TurboKV* does not only skip the excess forwarding step but also removes the load from the storage nodes from being the request coordinator, which reduces tail latencies at the storage nodes.

Figures 3.15(a), and 3.15(b) show that *TurboKV* reduces the scan latency by range of 13 - 23% for uniform workloads, and by 16 - 22% for skewed workloads when compared with the server-driven coordination (S-Coord). But, when compared with the ideal client-driven coordination (C-Coord), *TurboKV* increases the latency of the scan operation by range of 2 - 15% for uniform and skewed workloads, because of the latency introduced inside the switch from packet circulation and cloning to divide the requested range when it spans multiple storage nodes.

3.8.3 Comparison with Pegasus - one Rack

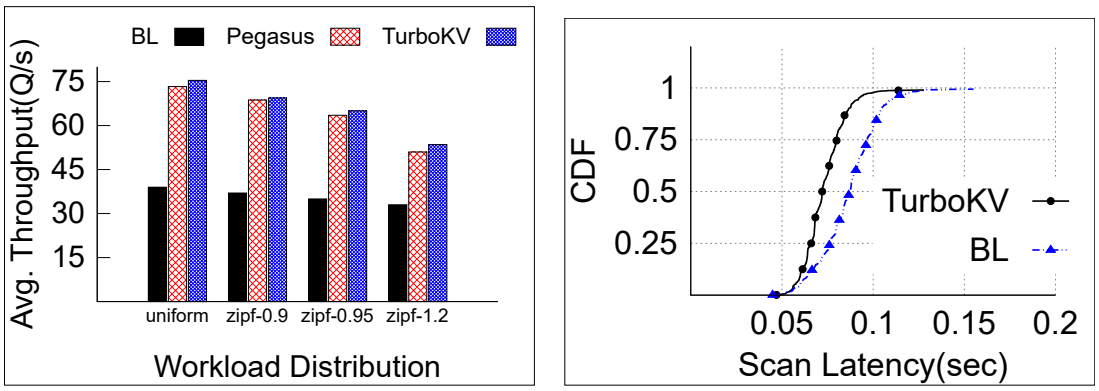
We compared the performance of *TurboKV* and Pegasus [35] on a rack scale topology, where one programmable switch connects 4 storage nodes and one client. We compiled both *TurboKV* and Pegasus on the simple software switch BMV2 running on Mininet. We used the BMV2 version of Pegasus from their repository on [50].

Figure 3.16(a) shows the system throughput under different skewness values with mixed read/write queries (50%read, 50%write). We compare *TurboKV*, the server-driven coordination (S-Coord) as the baseline (BL) and Pegasus. As shown in Figure 3.16(a), *TurboKV* performs nearly the same as Pegasus under different workload distributions. Both of *TurboKV* and Pegasus outperform the BL with minimum of 62% and maximum of 93%. This result is because *TurboKV* manages the directory information for all key-value pairs (total key-span) in the switch data plane and uses the key-based routing to deliver the requests to the target storage nodes directly without any prior knowledge from the client about these storage nodes. *TurboKV* also can handle the directory for all types of key-value stores, when data is range partitioned or hash partitioned. It also supports the easy scaling out of key-value stores for both types of partitioning techniques. On the other hand, Pegasus stores only the routing information for $O(nlogn)$ of the most popular keys and uses this information to route the requests to the least loaded storage node. Other keys are mapped to a home server using a fixed algorithm. This home server will be responsible for request coordination (server-driven coordination). Unfortunately, this method introduces extra forwarding steps if the home server does not have the data.

Figure 3.16(b) shows the CDF for scan latency for the skewed zipf-1.2 workload for a rack-scale configuration. *Pegasus is not shown as it does not support range scan operation.* Figure 3.16(b) shows that *TurboKV* also outperforms the BL on the rack scale configuration and reduces the latency of the scan operation.

3.9 Related Work

Distributed Key-value Stores. Key-value storage is widely used to support lots of large-scale applications. Some key-value stores, e.g., Redis [9], RAMCloud [10], and memcached [51], manage data in DRAM for faster data access. Other key-value stores, e.g., Dynamo [8], Cassendra [29], LevelDB [11] and RocksDB [12] are presistent key-value stores which save data on presistent storage devices, while other key-value stores, e.g., AeroSpike [30] use hybrid storage(DRAM and SSD). Distributing data over several key-value store instances has been widely studied. Some systems use hash functions to distribute the data among storage nodes. Dynamo [8] and Cassandra [29] use consistent



(a) Throughput vs Skewness (b) Scan Latency(Pegasus doesn't support scan)

Figure 3.16: Pegasus and TurboKV Comparison - One Rack

hashing, while Redis [9] and Aerospike [30] use a hash function to distribute data into several hashing slots. Other key-value stores, e.g., LevelDB [11] and RocksDB [12], use alternate approach to distribute the data, they use the key range partitioning where keys are on Sorted String Tables. *TurboKV* supports hash partitioning of the data and range partitioning. Applications can decide the way to partition the data and the corresponding directory information will be stored in switches' data plane.

To achieve durability and high availability, data partitions are replicated over several storage nodes. Dynamo, Cassendra, Aerospike, and Redis replicate the data partitions over several storage nodes and save this information on a mapping table which is replicated also on all storage nodes. *TurboKV* also replicates the data partitions over several storage nodes and follows the chain replication model to guarantee strong consistency. It stores the chain of each partition on the switch data plane. All existing key-value stores use either client-based coordination [8, 30], server-based coordination [8, 9, 29] or a single elected node coordination [10] to deliver requests from clients to storage nodes. *TurboKV* uses in-switch coordination with the key-based routing protocol to route requests from clients to storage nodes directly.

Hardware Acceleration. Lots of work used hardware to speed up the performance of the distributed systems. NetPaxos [20,21] implements Paxos on switches. NetCache [22] implements a rack-scale on-switch cache, while DistCache [24] scales up the NetCache design to multiple racks in the data center. NetChain [23] uses the switches to implement

in-network key-value store, but it is bounded by the limited storage in these switches. Concordia [36] also uses the programmable switches to scale up the performance of distributed shared memory systems. SwitchKV [52] uses the OpenFlow switches to save a forwarding rule for each cached key-value pair to route the request to the right caching node. Pegasus [35] uses the switches along with the selective replication approach for load balancing, it maintains a coherent in-network directory for the most popular key-value pairs, and distributes the load between the storage nodes that have these replicated items. *TurboKV* uses the programmable switches to solve the partition management problem; the switches act as the request coordination nodes that save the partition management information along with the key-based-routing protocol to route the request to the target storage nodes. In *TurboKV*, key-value pairs are saved on storage nodes which makes it not limited to applications of small data sizes.

Other examples that use hardware to scale up the perfromance include, but not limited to, JoiNS [53] which uses the OpenFlow switches to prioritize I/O packets to meet their latency SLO, KVDirect [54] that uses programmable NIC and enable remote key-value access to the main host memory, iSwitch [25] which uses the switches to improve the performance of the distributed reinforcement learning, and Ibex [55] which supports advanced SQL offloading using FPGA.

3.10 Conclusion

In this chapter, we presented *TurboKV*; a novel distributed key-value store architecture that leverages the power and flexibility of the new programmable switches. *TurboKV* uses the in-switch coordination approach that utilizes the switches as partitions management nodes to store the key-value store partitions locations and replicas information along the path from clients to storage nodes. The programmable switches use key-based routing to route packets from clients to storage nodes. *TurboKV* decreases the query response time and improve system throughput. We believe that *TurboKV* can be deployed on the programmable switches currently integrated in the data center's network to improve the performance of distributed key-value stores.

Chapter 4

TransKV: A Networking Support for Transaction Processing in Distributed Key-value Stores [2]

4.1 Introduction

Big data has attracted lots of people's attention. Nowadays, enormous amount of data has been generated and analyzed for the benefit of society at a large scale. With this huge amount of generated data, data is distributed among several storage instances, accessed frequently, retrieved and processed by many applications to extract useful information. So, it is important to improve the data access performance when data is accessed from storage nodes through network.

Nowadays data is being generated by many different sources with un-unified structures, hence this data is often maintained in key-value storage, Which is widely used due to its efficiency in handling data in key-value format, and flexibility to scale out without significant database redesign. According to DB-Engines [56], key-value store is one of the most popular NoSQL databases which are broadly used as the storage engine for high-traffic websites and other high-performance content. Examples of popular key-value stores include Dynamo [8], RocksDB [12], Redis [9], Memcached [51]. These key-value store engines have been extensively used by different clients including Amazon, Facebook, Nokia and Samsung [57].

Some of the applications built on these key-value stores employ non-trivial concurrent transactions from multiple clients. Consequently, managing all of these concurrent transactions without adequate concurrency control creates significant problems for the application. For example, in Amazon's e-commerce platform, the shopping cart service processes tens of millions requests that come from over 3 million checkouts in a single day. Each of these requests represents a transaction, that should guarantee the different ACID (Atomicity, Consistency, Isolation and Durability) properties in order to reach a correct database state.

Unfortunately, the distributed architecture of key-value stores makes it difficult to implement the required ACID properties for supporting transactions. Implementing the transaction concepts has a negative effect on the main two targets of any key-value store: scalability and predictable perfromance, as shown in Figure 4.1. This effect is due to the complexity, locking, starvation introduced by transactions and the interference with other non-transaction operations. That is why, some key-value stores [29, 30, 58] omit the transaction concepts. However, other key-value stores [9, 59] support transaction concepts by introducing a transaction coordinator. The transaction coordinator is responsible for the coordination among the key-value storage nodes. Each storage node implements a concurrency control mechanism to decide whether to accept or reject a transaction, then the transaction coordinator aggregates these decisions from the participating nodes and decides whether to abort or accept the transaction before processing it. Unfortunately, this model introduces lots of communications and forwarding steps in key-value queries processing, which is usually carried out through network switches. These additional steps increase the response time of the key-value queries.

On the networking side, Software-defined Network (SDN) simplifies network devices by introducing a logically centralized controller (control plane) to manage simple programmable switches (data plane). Recently, the Programming Protocol-Independent Packet Processor (P4) [14] unleashes capabilities that give the freedom to create intelligent network nodes performing various functions. Thus, applications can accelerate their performance by offloading part of their computational tasks to these programmable switches to be executed in the network. Nowadays, programmable networks get a bigger foot in the data center doors. Google cloud started to use the programmable switches

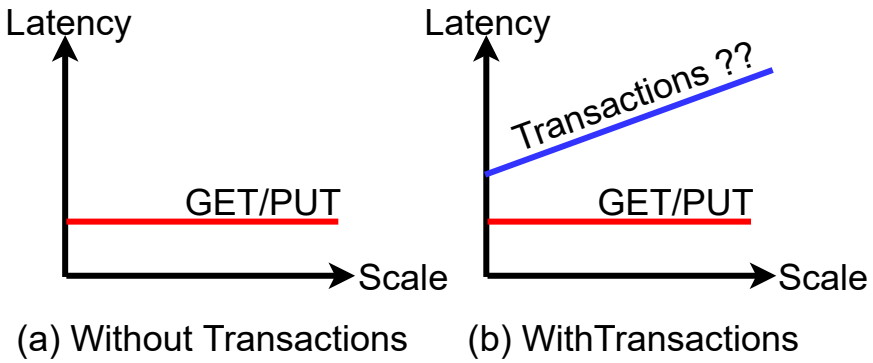


Figure 4.1: Performance of KV Stores w/o Transactions

with P4Runtime [15] to build and control their smart networks [16]. Some Internet Service Providers (ISPs), such as AT&T, have already integrated programmable switches in their networks [17]. These programmable switches are managed by the network administrators to adapt with the current network requirements and application needs.

Now, as we have control over both network and storage, it is time to think about how to improve data access performance when applications access storage through network. In this chapter, we propose TransKV: a network support for transaction processing using the programmable switches to further improve the latency of transactional key-value queries. We believe that network latency has a significant impact on the performance of transactions which have to be processed by the storage system in order to ensure serializability. TransKV utilizes the programmable switches as a concurrency control manager to execute the transaction processing logic in network. We are developing a variation of the Timestamp Ordering (TSO) [31] algorithm on the programmable switches. If a transaction can start processing according to the TSO logic, it is accepted and forwarded to the storage nodes. Otherwise, the transaction is aborted early by the programmable switches, and packets are routed back to the client.

TransKV adapts a hierarchical caching scheme [60–65] to distribute the hottest key-value pairs on the data plane of data center's switches, where higher level of cache contains the hottest key-value pairs, and the hotness of data decreases while going down in the hierarchy. TransKV provides a transactional support by injecting some information about the requested data in packet headers. The programmable switches use this information along with the timestamps saved for all cached key-value pairs to

decide whether to accept the submitted transaction and send it to the storage nodes for processing, or abort the submitted transaction directly from the network and send the packet back to the client.

TransKV utilizes the architecture of software-defined network [13, 40]. In our architecture, a logically centralized controller has a global view of the whole system [41]. This logically centralized controller manages the log of all transactions' history for failure recovery. It acts also as the transaction coordinator for the non-cached key-value pairs. It also updates the cache of each switch by the hottest key-value pairs periodically. Our Experimental evaluation based on our initial prototype shows that TransKV improves the throughput by up to 4X and reduces the latency by 35% on average

The remaining sections of this chapter are organized as follows. Quick introduction about the timestamp ordering algorithm (TSO) is discussed in Section 4.2. Section 4.3 provides the architecture of TransKV, while the detailed design of TransKV is presented in Section 4.4. TransKV implementation is discussed in Section 4.5, while Section 4.6 gives an experimental evidence and analysis of TransKV. Section 4.7 provides a short survey about the related work, and finally, our work is concluded in Section 4.8.

4.2 Background

4.2.1 Timestamp Ordering Concurrency Control

Transaction processing systems require high availability and fast response time for thousands of concurrent users. With the concurrent access of hundreds of users, concurrency control is needed to ensure the correct execution of users' transactions, when multiple transactions submitted by various users interfere with one another in a way that produces incorrect results. Timestamp Ordering (TSO) [31, 66] is a concurrency control protocol that guarantees serializability using transaction's timestamps to order transaction execution for an equivalent serial schedule. The idea for this timestamp ordering scheme is to order the transactions based on their timestamps, transaction start time, then the transaction processing system will only allow the transactions to be processed according to that timestamp ordering.

The timestamp ordering algorithm must ensure that, for each item accessed by conflicting operations in the schedule, the order in which the item is accessed does not

violate the timestamp order. To do this, the algorithm associates with each database item X two timestamp (TS) values: read timestamp ($R-TS(X)$) and write timestamp ($W-TS(X)$). The read timestamp of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X. The write timestamp of item X is the largest of all the timestamps of transactions that have successfully written item X. The system checks timestamps for every operation. If a transaction tries to read/write an object from the future, i.e., with greater read/write timestamp than the transaction's timestamp, it aborts and restarts. Timestamp ordering is different from any locking protocol; it determines serializability order of transactions based on their timestamps before execution without using any locks, and hence it is a deadlock-free protocol which make it more suitable for the programmable switches architecture.

4.3 TransKV Architecture Overview

TransKV is a networking support for transactions in distributed key-value stores that leverages the capability of programmable switches. TransKV utilizes the programmable switches as a transaction manager to coordinate between the submitted transactions. Figure 4.2 shows the architecture of TransKV within a data center, which consists of the programmable switches, controller, storage nodes, and clients. We are going to discuss the role of each component in this section.

Programmable Switches. Programmable switches are the essential component in our proposed system. We augment the programmable switches with a cache [22] to store the popular key-value pairs, and leverage match-action tables and registers in the programmable switches to design the in-switch transaction coordinator where the values and the timestamps information of cached key-value pairs will be stored. The programmable switch uses a variation of the TSO along with the stored information to decide whether the submitted transaction can start processing and be routed to the target storage node, or can be aborted directly from the network before reaching the storage node. Following this approach, the programmable switches act as transaction coordinator nodes that coordinate between the submitted transactions.

In addition to transaction coordination, each programmable switch has a query statistics module to provide the controller with statistics reports to enable it to update

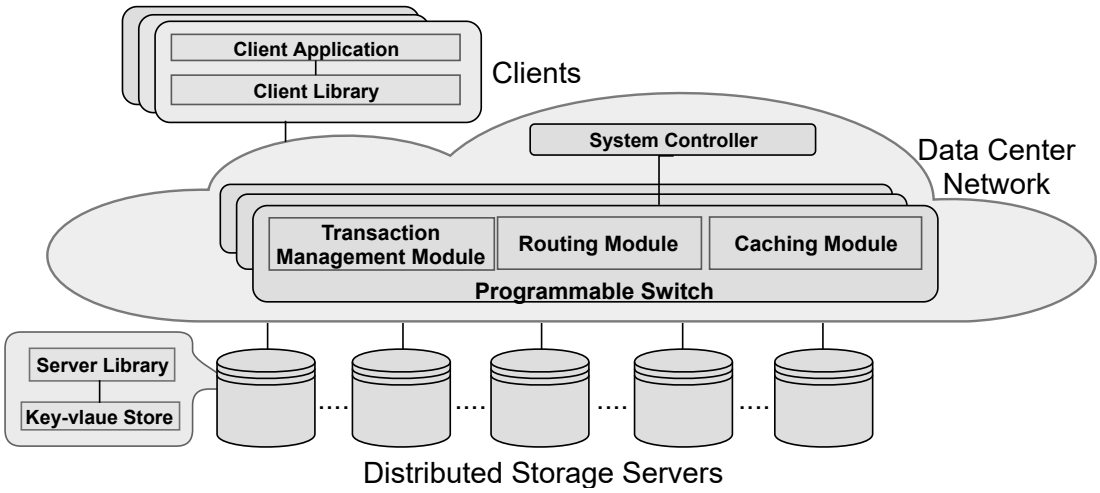


Figure 4.2: TransKV Architecture Overview

the cache with the popular key-value pairs. TransKV can distinguish between packets. Packets marked as TransKV packets, are processed by our system. Other packets are processed and routed using the standard L2/L3 protocols which make TransKV compatible with other network functions and protocols.

Controller. The controller is responsible for system reconfigurations including (a) log management for recovering from system failures, (b) transaction coordinator for non-cached key-value pairs, and (c) updating each switch's cache with the recent popular key-value pairs. The controller keeps track with all changes made by the transactions on a log. In case of any system failure, this log is done/undone in order to restore the system to a consistent state. Through the control plane, the controller also updates the match-action tables in the switches with the new popular key-value pairs. TransKV controller is an application controller that is different from the network controller in SDN, and it does not interfere with other network protocols or functions managed by the SDN controller. *Our controller only manages the transaction log and the cached key-value pairs. Both controllers can be co-located on the same server, or on different servers.*

Storage Nodes. They represent the location where the key-value pairs reside in the system. The key-value pairs are partitioned among these storage nodes. Each storage node runs a simple shim that is responsible for reforming TransKV query packets to

API calls for the key-value store. This layer makes it easy to integrate our design with existing key-value stores without any modifications to the storage layer.

Clients. TransKV provides a client library which can be integrated with the client applications to send TransKV packets through the network, and access the key-value store without any modifications to the application. Like other key-value stores such as LevelDB [11] and RocksDB [12], the library provides an interface for all key-value pair operations (PUT, GET, DELETE) that is responsible for constructing the TransKV packets and translates the reply back to the application.

4.4 TransKV Design

The data plane provides on-switch cache and transaction coordination model for the key-value stores to handle concurrency control earlier in network. In this model, all timestamps for cached key-value pairs are stored on switches, and are used for transaction coordination. Figure 4.3 represents the whole pipeline that the packet traverses inside the switch before being forwarded to the storage node for processing, or aborted by the switch and forwarded back to the client. As shown in Figure 4.3, when the packet arrives at the switch, it is parsed with the parser to extract the headers. If the packet contains TransKV header, it is processed with our TransKV modules in the ingress pipeline, and then forwarded to the next hop. In this section, we discuss how the switch supports these functions.

4.4.1 Network Protocol Design

Packet Format. Figure 4.4 shows the format of TransKV request packet sent from the clients to the storage nodes. The programmable switches use a reserved port number in the TCP/UDP header to identify TransKV packets, and lookup the cache for non-transactional packets, or execute the transaction management process for the transactional packets. Other switches in the network do not need to understand the TransKV header, and treat all packets as normal IP packets. The TransKV header consists of three main fields: `T-TS`, `TLength`, and a list of `OP`. The `T-TS` is a 4-byte field which stands for the submitted transaction timestamp. A packet with `T-TS = 0` represents a

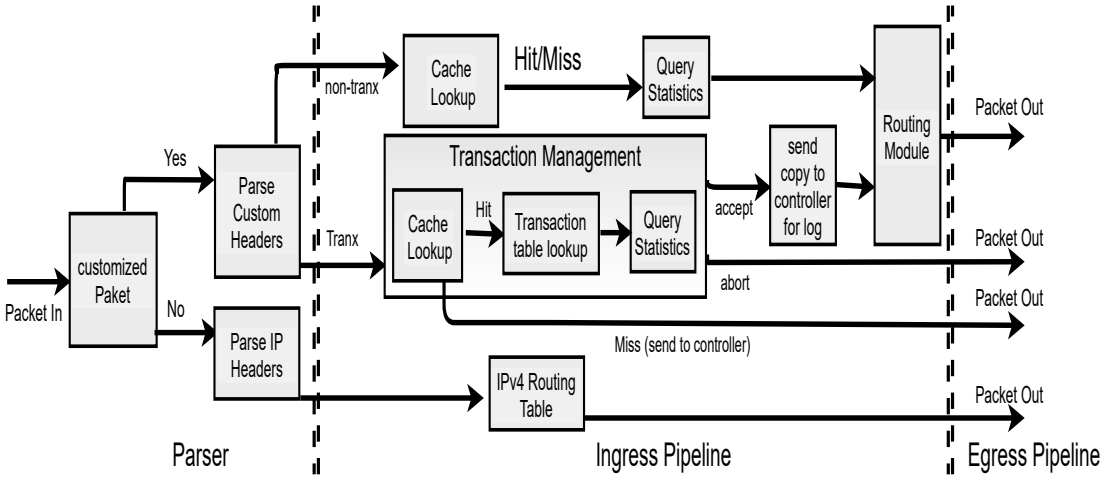


Figure 4.3: Logical View of TransKV Data Plane Pipeline

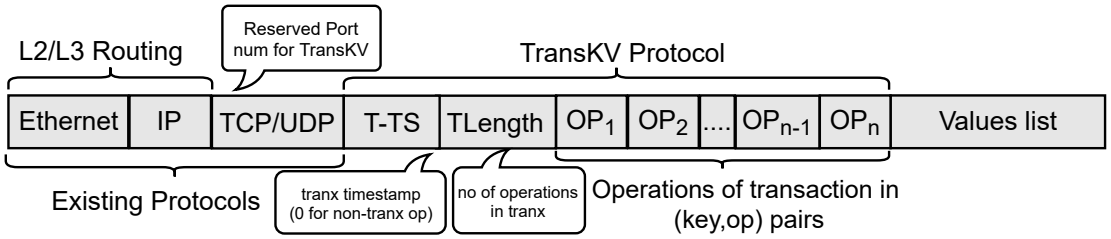


Figure 4.4: TransKV Packet Format

non-transactional operation. `TLength` stores the number of key-value operations on the submitted transaction. The `OP` list is a list of (key,op) pairs of length equals to `TLength`, each pair represents a 4-bit operation followed by a requested key.

After the packet is processed by the programmable switch, the switch either accepts the packet and sends it to the storage nodes for processing, or aborts the transaction and sends the reply back to the client. The reply packet is a standard IP packet, and the result is added to the packet payload.

Network Routing. TransKV uses existing routing protocols to forward clients' packets through the network. For a TransKV packet, based on the information of data location, the client appropriately sets the Ethernet and IP headers and sends the packet to the storage server where data resides. The programmable switches, placed on the path from the clients to the storage clusters, process TransKV packets. If the packet represents a

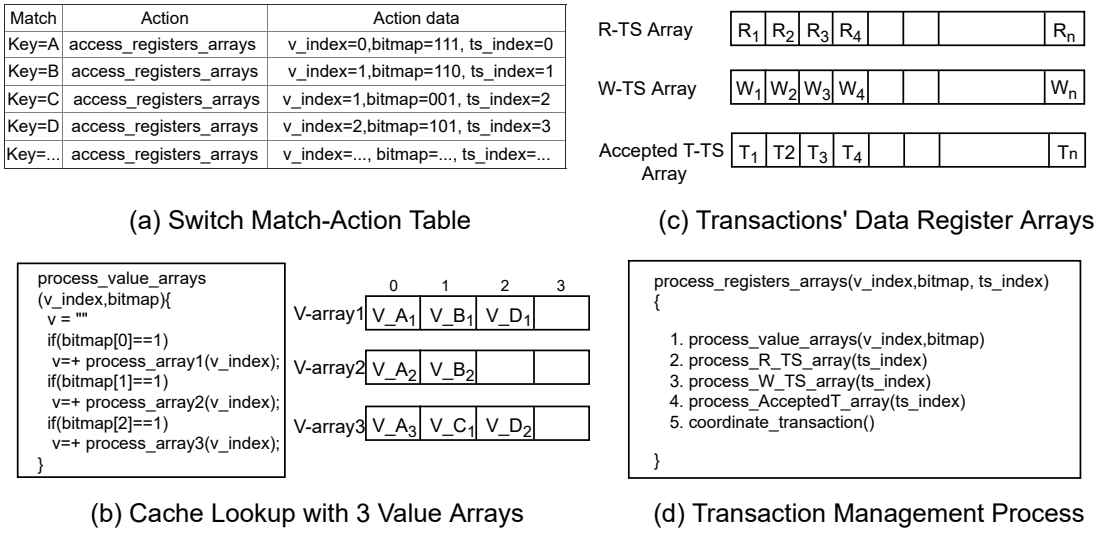


Figure 4.5: TransKV Design inside Switch Data Plane

transaction, the switch either accepts the packet and sends it to the storage nodes for processing or aborts the transaction and sends the reply back to the client. If the packet represents a non-transactional operation, the switch attaches the value to the packet, in case of cache hit, and sends the reply back to the client, or sends the packet to the storage nodes if the requested item is not in cache. Other switches simply forward packets based on the destination MAC/IP address according to the L2/L3 routing protocol. In this way, TransKV can coexist with other network protocols.

4.4.2 On-Switch Cache

TransKV adopts on switch cache, where hot key-value pairs are stored on switch match-action table and switch registers. There are two types of match-action tables inside the switches: the routing match-action table and the key-value cache. In this section we will discuss the design of the key-value cache. The key-value cache design is shown in Figure 4.5(a). Each record in the table consists of three parts: *match*, *action* and *action data*. The *match* represents the value that we match the requested key against, TransKV uses the exact-match to match a requested key against a record in the match-action table. The *action* represents the transaction management process that will be executed when a requested key matched a record in the match-action table. The *action*

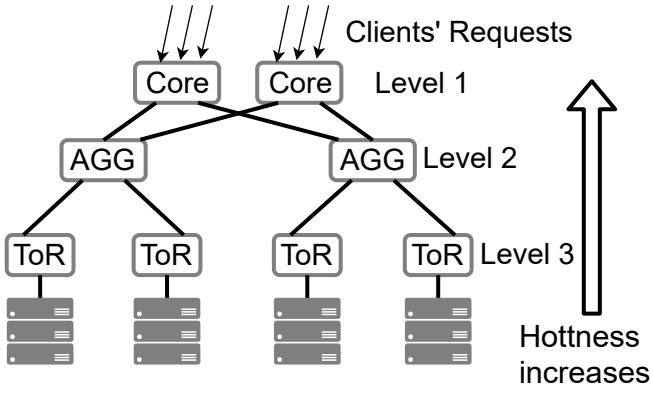


Figure 4.6: Hierarchical Caching inside Data center's Switches

data consists of three parts: `v_index`, `bitmap` and `ts_index`. `v_index` and `bitmap` are used as in [22] to support storing of variable length key-value pairs in the switch's registers.

We use a set of register arrays to store the values of the key-value pairs. Each register array consists of fixed-length slots. The value associated to a specific key is divided between these register arrays, and each part of the value is stored at the same index (`v_index`) in all value arrays. the `bitmap` is a bit-vector of length equal to the number of register arrays used to store values. A 1 in position x in the bitmap indicates that the value array number x has a part of the value associated to the requested key, and a 0 in position x in the bitmap indicates that the value array number x doesn't contribute to the requested key's value. Figure 4.5(b) shows an example of 3 registers cache and how we use the `v_index` and `bitmap` to retrieve the value of a requested key. `ts_index` represents the index of the timestamps associated to a key-value pair in the switch registers. The timestamps of a record are stored at the same index in all registers. There are three registers used to store the timestamps as shown in Figure 4.5(c): read timestamp register, write timestamp register and submitted transaction timestamp register. The usage of each of these registers will be discussed in Section 4.4.3.

TransKV uses a hierarchical caching approach based on the Least Frequently Used (LFU) eviction policy [67] [68] as shown in Figure 4.6, the total cache size equals to $n \times m$, where n is the number of switches in the data center network and m is the size of

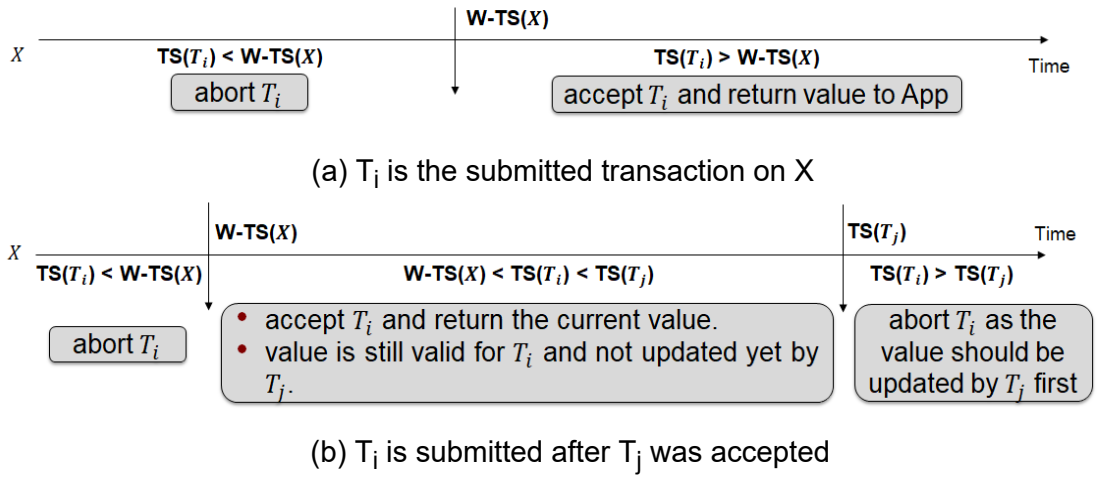


Figure 4.7: Acceptance or abortion of Read Operation

each switch cache. Each key-value pair is cached only once on the whole cache (except for the core switches that hold the same cached key-value pairs), where the hottest key-value pairs are cached on top level (level 1) where the client requests are submitted, hot key-value pairs are cached on the middle level (level 2), and warm key-value pairs are cached on the bottom level (level 3). The controller decides which key-value pairs reside in each level based on the number of requests submitted on each key-value pair in a specific time window.

4.4.3 On-Switch Transaction Management

TransKV acts as a transaction manager for the cached key-value pairs. Figure 4.5(d) shows the transaction management process. A variation of Timestamp Ordering (TSO) concurrency control protocol is implemented on the programmable switches. TSO is chosen because it is a deadlock free approach; there is no locking mechanism used on the records, and hence there is no checking in the dependency graph for cycles, which makes it suitable for the programmable switch's match-action pipeline. For each cached key-value pair, the read timestamp $R-TS$, the write timestamp $W-TS$, and the timestamp of the current accepted transaction $T-TS$ are stored on the switch registers as shown in Figure 4.5(c).

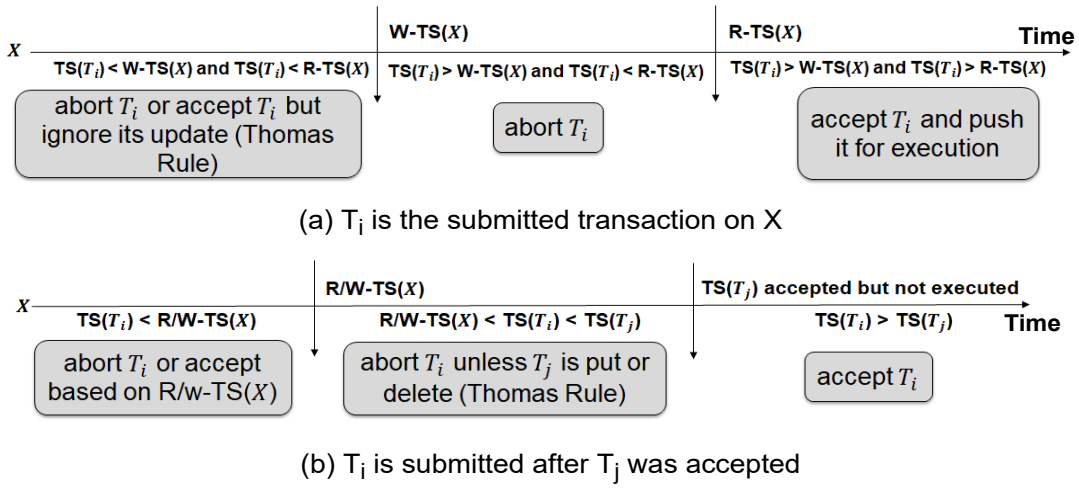


Figure 4.8: Acceptance or abortion of write Operation

Read Operation within a transaction. If there is a submitted transaction T_i with timestamp $TS(T_i)$ submitted on one of the cached key-value pairs X, where T_i contains a read operation, the switch will fetch X's corresponding timestamps from the registers $R-TS(X)$, and $W-TS(X)$. Then, the switch compares $TS(T_i)$ with X's timestamps and decide whether to accept or abort the operation as shown in Figure 4.7(a). If the timestamp of the submitted transaction ($TS(T_i)$) is less than the current write timestamp of X ($W-TS(X)$), then the transaction T_i will be aborted because it tries to read an item that was modified by an older transaction (T_i tries to read a value from the future). If the timestamp of the submitted transaction ($TS(T_i)$) is greater than the current write timestamp of X ($W-TS(X)$), then the transaction T_i reads the item successfully and return the value to the client. After reading X, the read timestamp of X ($R-TS(X)$) will be updated with max ($TS(T_i)$, $R-TS(X)$).

If there is a previously accepted transaction T_j on X ($T-TS > 0$) and the value of X isn't updated yet by T_j, the switch will fetch $TS(T_j)$ from the register of submitted transaction, and decide whether to accept or abort the operation as shown in Figure 4.7(b). T_i will be accepted only if the $W-TS(X) < TS(T_i) < TS(T_j)$, because the current value of X is still valid for T_i and not updated yet by T_j.

Write Operation within a transaction. If there is a submitted transaction T_i with timestamp $TS(T_i)$ submitted on one of the cached key-value pairs X, where T_i contains a

write operation, the switch will fetch X's corresponding timestamps. Then, the switch compares these timestamps and decide whether to accept or abort the operation as shown in Figure 4.8(a). The transactionT_i will only be accepted if the timestamp of the transaction is greater than $R - TS(X)$ and $W - TS(X)$. If there is a previously accepted transaction T_j on X $(T - TS > 0)$ and the value of X isn't updated yet by T_j, the switch will fetch $TS(T_j)$ from the register of submitted transaction, and decide whether to accept or abort the operation as shown in Figure 4.8(b). T_i will be accepted also if the $TS(T_i) > TS(T_j)$, because the current value of X should be updated first by T_j then by T_i.

After the accepted transaction is processed by the storage node, the storage node sends the transaction back to the switch with a committed status. The switch uses the value in the packet of committed transaction to update the value stored on the switch. So the updated value will be available to other transactions submitted on the same key-value pair.

Transaction that contains multiple key-value operations (`Tlength > 1`) will be accepted if all its key-value operations are accepted, and it will be aborted otherwise. The transaction could pass by several switches in the path to the target storage nodes before it is accepted or aborted; as the requested key-value pairs may be cached on different switches.

4.4.4 Query Statistics

In TransKV, the data plane has a query statistics module to provide query statistics reports to the controller about the popularity of key-value pairs. Thus, through control plane, the controller updates the cache on each switch with the most popular key-value pairs. As shown in Figure 4.9, the switch's data plane maintains a per-key counter for each key in the match-action table. Upon each hit for a key, its corresponding counter is incremented by one. The switch also will notify the controller about each cache miss, so the controller can estimate the popularity of non-cached items. The controller receives reports periodically from the data plane including these statistics, and resets these counters periodically. Based on the received statistics, the controller updates the cache and all registers with the new popular items.

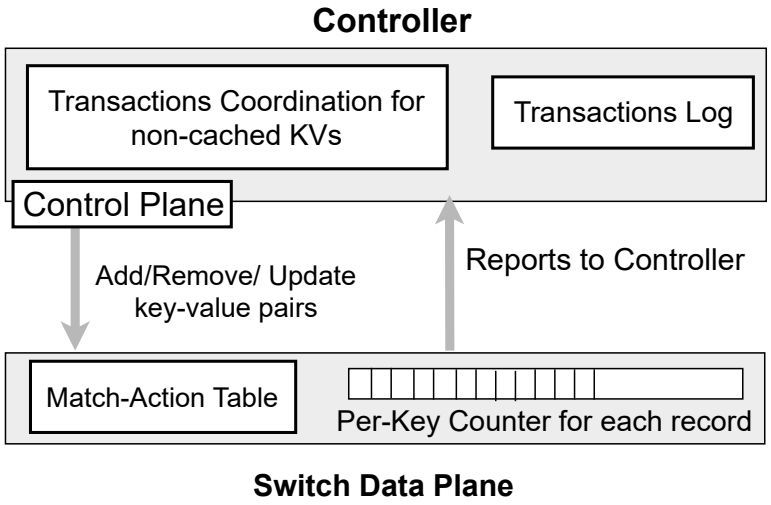


Figure 4.9: Logical View of The Controller and Query Statistics

4.4.5 Transaction Log Management

One of the main transaction concepts is consistency, which means that data must be in a consistent state when the transaction starts, and when it ends. If a failure happens during the processing of any transaction, all changes made by that failed transaction must be undone to guarantee data consistency. That is why TransKV controller stores a log of all submitted transactions' history, this log is used to redo/undo the changes made by committed/failed transaction. When a transaction is submitted and reaches the programmable switch, the switch sends a copy of this transaction to the controller, then the controller fetches the old data from key-value stores that is related to the submitted transaction, and finally controller creates a record for that transaction with the after and before images, and append this record to the recovery log. If a failure happens, the controller rolls back all the changes made by any uncommitted transaction, rolls back also any transaction that read a value written by the failed transaction (cascading rollback), and restore the database to a consistent state.

4.4.6 Transaction Management for non-cached KV Pairs

The programmable switches have a limited on-chip capacity, so all key-value pairs can not be cached on the switches, and hence switches can only cache the most popular

key-value pairs, and manage the transactions targeting these cached data. For other non-cached key-value pairs, the controller will take the transaction coordination role. The controller will act as the transaction coordinator described in [9,59]. The controller checks with each storage node, participating in the transaction. Each storage node sends its decision of whether to accept or reject the transaction, then the controller aggregates these decisions and sends the final decision back to the client. Because of the 80/20 rule in data science, we can see that only 20% of data is accessed 80% of the time and vise versa. By applying this rule on TransKV, nearly 80% of submitted transactions will be managed by the switches and 20% will take the normal path of transaction coordination via the controller, so the amortized response time for the transactions will be improved.

4.5 Implementation

We have implemented a prototype of TransKV, including all switch data plane and control plane features, described in Section 4.4. We have also implemented the client and server libraries that interface with applications and storage nodes, respectively, as described in Section 4.3. The switch data plane is written in P4, and due to lack of real hardware, it is compiled to the simple software switch BMV2 [42] running on Mininet [47]. The key size of the key-value pair is 16 bytes with total key range spans from 0 to 2^{128}. The cache lookup table has 64K entries. We used 3 registers for storing the value, each register has a 64K 16-byte slots with value granularity of 16 byte and up to 48 bytes. We also used 4 registers, each of them has 64K 4-byte slots: 3 of them for storing the timestamps of all cached key-value pairs, and the other register is used to count the access requests of cached key-value-pairs for query statistics module. The controller is able to update/read the values of these registers through the control plane. It also can add/remove key-value pairs to/from the caching table. The controller is written in Python and can update the switch data plane through the switch driver by using the Thrift API generated by the P4 compiler. The total on-chip used memory is around 5MB leaving enough space for processing other network operations.

The client and server libraries are written in Python using Scapy [48] for packet manipulation. The client can translate a YCSB [38] workload with different distributions and mixed key-value operations into TransKV packets and send them through the

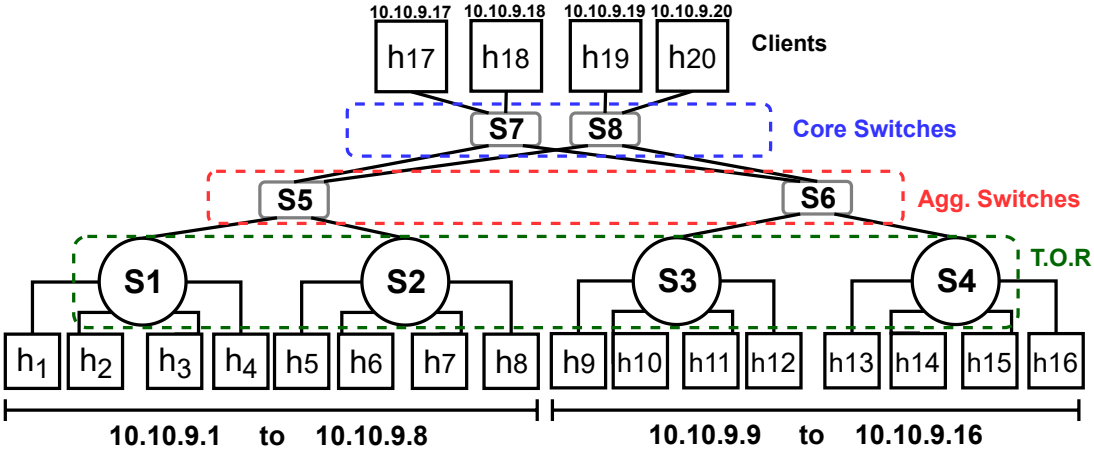


Figure 4.10: TransKV Experiment Topology

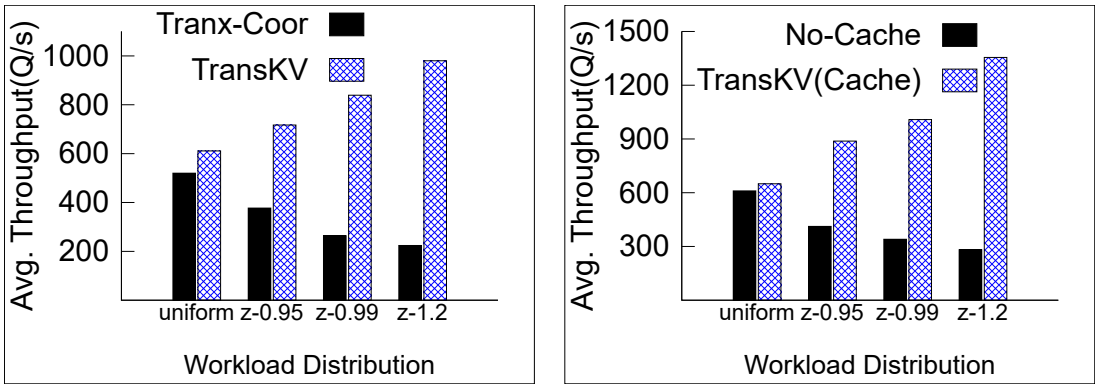
network. We used Plyvel [49] which is a Python interface for levelDB [11] as the storage agent. The server library translates TransKV packets into Plyvel format and connects to levelDB to perform the key-value pair operations.

4.6 Performance Evaluation

This section provides the experimental results of TransKV. We show the performance improvement of TransKV on the key-value operations latency and system throughput.

Experimental Setup. Our experiments consist of eight simple software switches BMV2 [42] running on Mininet (2 Core switches, 2 aggregation switches and 4 ToR switches). The switches are connected together as shown in Figure 4.10. Each ToR switch is connected to 4 storage nodes with total of 16 storage nodes. Each core switch is connected to 2 clients with total of 4 clients. Each of the clients runs the client library and generates the key-value transactional and non-transactional operations. Each storage node runs the server library and uses LevelDB as the storage agent. The data is range-partitioned over the storage nodes, where each storage node is responsible for handling part of the total key span.

Comparison. We compared our in-switch transaction management model (TransKV) with the transaction coordinator model (we refer to it as Tranx-Coor) shown in [59].

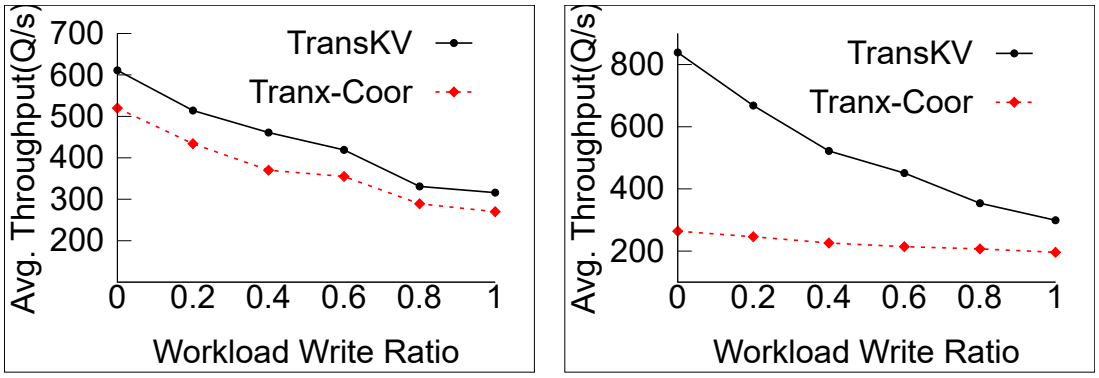


(a) Transactional Operations (b) Non Transactional Operations

Figure 4.11: TransKV Throughput vs Skewness - Read Only

In TransKV, the hottest key-value pairs are stored on the switch cache. Any submitted transaction on one of the cached key-value pairs is managed by the switch. Other non-cached key-value pairs are managed by TransKV controller which acts as the transaction coordinator. In Tranx-Coor, The transaction coordinator is responsible for the coordination among the key-value storage nodes. Each storage node implements the TSO concurrency control mechanism to decide whether to accept or reject a transaction, then the transaction coordinator aggregates these decisions from the participating nodes and decides whether to abort or accept the transaction before processing it.

Workloads. We use both uniform and skewed workloads to measure the performance of TransKV under different workloads. The skewed workloads follow Zipf distribution with different skewness parameters (0.95, 0.99, 1.2). These workloads are generated using YCSB [38] basic database with 16 byte key size and 48 byte value size. The generated data is stored into files, then parsed by the client library to convert them into TransKV packets. We generate different types of workloads: transactional and non transactional read-only workload, transactional write-only workload and transactional mixed workload with multiple write ratios.



(a) Uniform (b) Zipf-0.99

Figure 4.12: TransKV Throughput with Different Write Ratios

4.6.1 Effect on System Throughput

Impact of Read-only Workloads. Figure 4.11(a) shows the system throughput under different skewness parameters with transactional read-only workload. We compare TransKV vs Tranx-Coor. As shown in Figure 4.11(a), TransKV outperforms Tranx-Coor by minimum of 1.2X in case of uniform workload and by maximum of 4.4X in case of skewed workload. This result is because TransKV manages the transactions for the hot key-value pairs directly in the switch data plane. It decides using the TSO whether to accept or reject the transaction, and eliminates any excessive communication steps between the transaction coordinator and the storage nodes for decisions aggregation in case of Tranx-Coor. Moreover, the key-value pairs are stored on the switch data plane, they are retrieved directly from the switch without the need to go to the storage node to fetch the value. Also, when the skewness parameter increases, the throughput of the system increases; more hits occur on the switch cache ,and small amount of key-value pair misses are coordinated through TransKV controller.

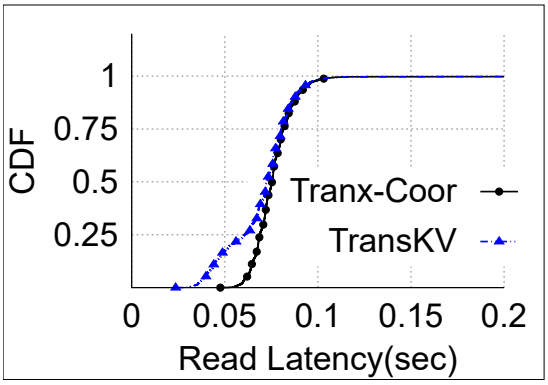
Figure 4.11(b) shows the system throughput under different skewness parameters for the non-transactional read-only workload. We compare the performance of TransKV when it caches the hottest key-value pairs in the switches without performing the TSO logic vs the performance of accessing the key-value pairs normally from the storage nodes without any caching on the switch side (No Cache). As shown in Figure 4.11(b), TransKV also outperforms the No Cache approach by 7% in case of uniform workload

and maximum of 5X in the skewed workload. This result is because frequent requests to hot data over cold data lead to load imbalance among storage nodes; some nodes are heavily congested while others become under-utilized. This results in a performance degradation of the whole system and a high tail latency. But using caching on the switch side absorbs the hot key-value queries and increase the throughput of the whole system. So we can conclude from Figure 4.11(a) and Figure 4.11(b) that TransKV improves the throughput of the key-value storage for both the transactional and non-transactional key-value operations.

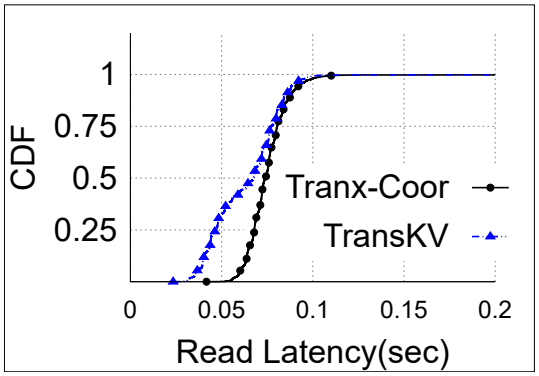
Impact of Write Ratio. Figure 4.12(a) and Figure 4.12(b) show the system throughput under uniform and skewed workload with varying the workload write ratio for transactional Operations. As shown in Figure 4.12(a) and Figure 4.12(b), TransKV outperforms the transaction coordinator (Tranx-Coor) for both the unifrom and skewed workload by minimum of15% and maximum of 25% in case of uniform workload and by minimum of 1.5X and maximum of 3X in case of skewed workload. This is because the management of the concurrency control logic in switch data plane and the elimination of excessive communication steps between the transaction coordinator and the storage nodes. We can see also from the same figures that as the workload write ratio increases, the throughput decreases. This result is because write requests can't be completed without writing the value in the storage node, so that other requests to the same Key-value pair can see the latest update. So when the write ratio increases, higher percentage of the requests will travel longer path to reach the storage node and persist the value which reduces the throughput of the system.

4.6.2 Effect on Key-value Transactions Latency

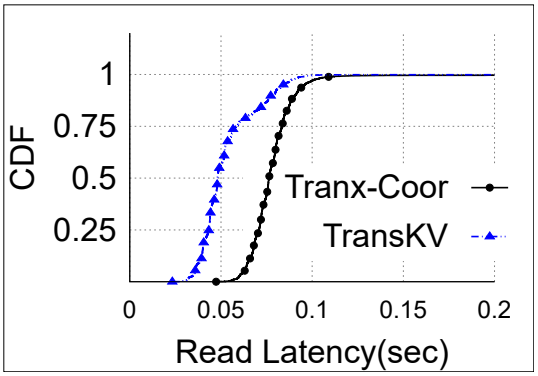
Figures 4.13(a), 4.13(b) and 4.13(c) show the Cumulative Distribution Function (CDF) of read transactions latency on key-value pairs under uniform, Zipf-0.99 and Zipf-1.2 workloads, respectively, for TransKV and Tranx-Coor. The analysis of these three figures is shown in Table 4.1. Figure 4.13(a) shows that TransKVperforms the same as the Tranx-Coor with very small latency improvement equals to 8% on average for the uniform workload; as the effect of caching significantly degrades when the data is uniformly accessed, small amount of key-value transactions will be managed by the



(a) Uniform



(b) Zipf-0.99



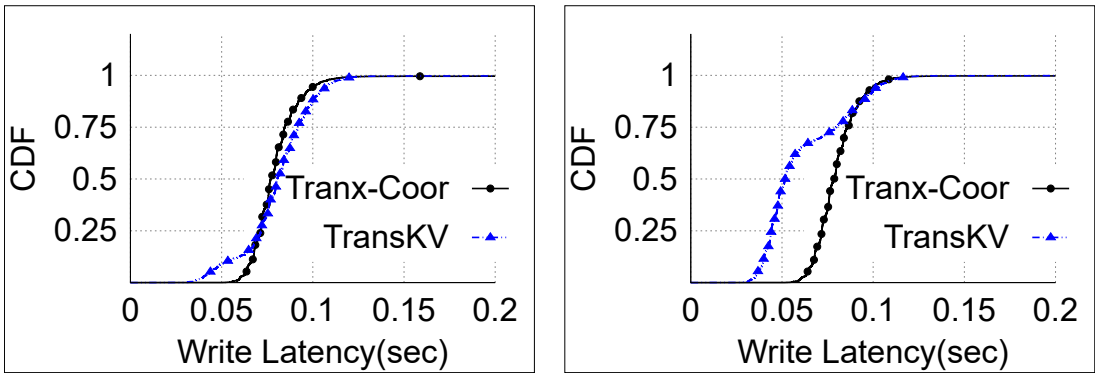
(c) Zipf-1.2

Figure 4.13: TransKV Effect on Read Latency

switch and other transactions will be managed by TransKVcontroller, which is the normal transaction coordinator path as well in Tranx-Coor. When the skewness of data increases, the caching effect goes in action with more hits in the switch cache. This effect makes TransKV outperforms Tranx-Coor with 15% and 32% on average in the case of zipf-0.99 workload and zipf-1.2 workload as shown in Figures 4.13(b) and 4.13(c), respectively; as TransKV manages larger number of transactions using TSO logic in the switch data plane and decides directly whether to accept or abort them. This approach makes TransKV avoids the excessive communications introduced by Tranx-Coor. Moreover, for read transactions, the transaction can retrieve the value directly from the switch cache without the need to go to the storage node to retrieve it.

	Uniform			Zipf-0.99			Zipf-1.2		
	Mean	50 Percentile	99 Percentile	Mean	50 Percentile	99 Percentile	Mean	50 Percentile	99 Percentile
TransKV	0.07061354	0.072377563	0.104059825	0.063888747	0.067531943	0.099380732	0.053317506	0.048128009	0.095414464
Tranx-Coor	0.076945137	0.075366497	0.104418943	0.075576334	0.074002504	0.104888678	0.078707565	0.076314449	0.111776471

Table 4.1: Read Transactions Latency Analysis Under Different Workloads



(a) Uniform (b) Zipf-1.2

Figure 4.14: TransKV Effect on Write Latency

Figures 4.14(a) and 4.14(b) show the CDF of write transactions latency on key-value pairs under uniform and Zipf-1.2 workloads for TransKV and Tranx-Coor. The analysis of these two figures is shown in Table 4.2. Similar to our observation from the read latency results, TransKV has no improvement in case of uniform workload, but TransKV outperforms Tranx-Coor when the data skewness increases as shown in Figure 4.14(b); this is because hit ratio in switch data plane increases, and TransKV manages the transactions of the cached key-value pairs directly in the switch. TransKV reduces the latency by 24% on average and by 34% for the 50^{th} percentile. The latency improvement in write transactions is less than the read transactions, because the write transactions need to reach the storage node to update the value and make it accessible for other transactions.

	Uniform			Zipf-1.2		
	Mean	50 Percentile	99 Percentile	Mean	50 Percentile	99 Percentile
TransKV	0.081189324	0.080517054	0.120302806	0.0614705	0.051731467	0.11642298
Tranx-Coor	0.080760114	0.076931477	0.116496089	0.080588034	0.078615069	0.114953876

Table 4.2: Write Transactions Latency Analysis Under Different Workloads

4.7 Related Work

Transactions on Distributed Key-value Stores and NoSQL systems. Key-value storage is widely used to support lots of large-scale applications. Some key-value stores manage data in DRAM for faster data access [9, 10, 51]. Other key-value stores are presistent key-value stores which save data on presistent storage devices [8, 11, 12, 29], while other key-value stores use hybrid storage(DRAM and SSD) [30].

Most key-value stores and NoSQL Systems like BigTable [58], Dynamo [8], PNUTS [69], MongoDB [70], CouchDB [71], and Cassandra [29] omit the transactions because of the negative effect on these systems' performance, but other key-value stores like Amzon DynamoDB [59] and Redis [9] use the concept of transaction coordinator to coordiante between transactions with seperate concurrency control mechanism implemented on the storage nodes to solve this problem. FoundationDB [72] provides strict serializable ACID transactions on a scalable key-value store by a lock-free concurrency control combining Multi Versions Concurrency Control (MVCC) and Optimistic Concurrency Control (OCC). It uses a timestamp for guarteeing the serial order of transactions. This timestamp is generated by a Sequencer. FoundationDB also uses resolvers to resolve conflicts based on the OCC. Omid [73,74] layered transactional APIs atop key value stores with snapshot isolation. Cherry Garcia [75,76] enables multi-item transactions with snapshot isolation across multiple heterogeneous data stores, it uses a client-coordinated transaction commitment protocol that does not rely on a central coordinating infrastructure. Wrap [77], a transactional system over key-value store, uses a protocol called acyclic transactions for providing serializable transactions on top of a sharded data store. It allows multiple transactions to prepare and commit simultaneously. TransKV supports transactions in distributed key-value stores using the programmable switches without any extra communications or forwarding steps introduced by the transaction coordinator, conflict resolver or other intermediate servers between the client and storage servers.

Hardware Acceleration. Emerging hardware technologies are used to speed up the performance of the distributed systems. Some distributed systems use the programmable switches to improve their performance [78] like: [22, 24, 35] that use the programmable switches to balance the load among storage nodes, NetChain [23] that

uses the switches to implement in-network key-value store, iSwitch [25] which uses the switches to improve the performance of the distributed reinforcement learning, JoiNS [53] which uses the OpenFlow switches to prioritize I/O packets to meet their latency SLO, Concordia [36], a distributed shared memory that use the programmable switches for in-network cache coherence and finally Gotthard [79] that uses the optimistic concurrency control along with the programmable switches to cache some transaction results, and based on that cached history, it aborts some transactions that are likely to be aborted by the storage server. Gotthard operates only on a single storage server and single switch. TransKV uses the TSO for transaction processing on the switches. It caches the hot key-value pairs on switch's data plane to execute the TSO logic and accept or reject transactions directly from network. Moreover, TransKV is scalable to the data center network with multiple switches and distributed Key-value nodes.

4.8 Conclusion

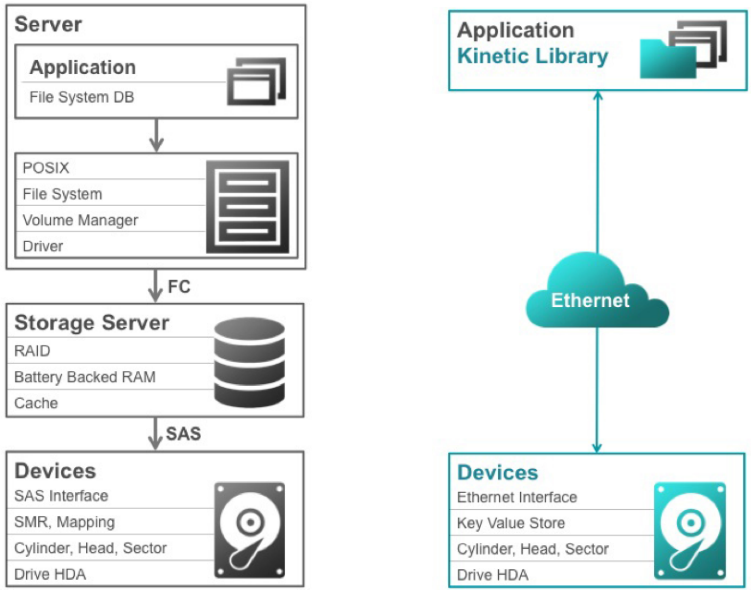
In this paper, we presented TransKV: a networking support for transaction processing in distributed key-value stores, that leverages the power and flexibility of the new programmable switches to act as a transaction manager. TransKV switches coordinate between the submitted transactions on the key-value pairs that are cached on the switch data plane, while TransKV controller manages the log for restoring the system to a consistent state after transactions failure. TransKV also takes the benefit of the data center's network structure to design a hierarchical caching mechanism on the switches. We believe that TransKV can be deployed on the programmable switches currently integrated in the data center's network to improve the performance of distributed key-value stores.

Chapter 5

Key-value Pairs Allocation Strategy for Kinetic Drives [3]

5.1 Introduction

The demand of achieving high performance in processing enormous amounts of digital unstructured data opens the door for the NoSQL databases, that provide flexibility and high performance over the traditional relational databases. NoSQL databases eliminate some performance bottlenecks introduced by the traditional relational databases. Moreover, the nature of nowadays data makes the default relational databases (RDBs) to not be a suitable option to process and store this data [80] [81]. Therefore, NoSQL databases become a competitive alternate to be considered. Key-value (KV) object storage, one of the NoSQL databases categories, is becoming extremely important; it treats the data as a single opaque collection which may have different fields for each record. In key-value storage, the record is represented by two attributes: the key which is used as an unique identifier to store, read, modify or delete the record and the value which is the data itself. The value is a variable-length object and can be used to store any type of data, such as files, database records, medical images, graphs, or multimedia. However, the length of keys to be stored is limited to a certain number of bytes, there is a less limitation on the values [82] [83]. This offers considerable flexibility because the application has full control to decide what gets stored in the value.



(a) Traditional Stack. (b) Kinetic Stack.

Figure 5.1: Traditional Storage Stack and Kinetic Drive Storage Stack [5]

Key-value object storage also follows modern concepts like object-oriented programming. It is a kind of software driven, as it communicates data in terms of objects rather than files or blocks. Moreover, key-value storage offers massive scalability. It can scale out easily on demand using commodity hardware without significant redesign of the database. To achieve this high scalability, key-value object storage omits some important features like consistency and also some analytical features like aggregate and join queries [80]. Also for structured data, optional values are not represented by placeholders as in most RDBs, which makes key-value stores often use far less memory to store the same database, which can lead to large performance gain in certain workloads. Many key-value storage systems are developed to support many large applications and websites like Amazon's Dynamo [8], Redis [9], RAMCloud [10], LevelDB [11], RocksDB [12] and project Voldemort [84].

On the storage Industry side, the Object Storage Devices (OSD) and active disks were introduced that can manipulate data in-terms of objects instead of file blocks. The Kinetic drive is an example of OSD and active disks introduced by SeaGate. The Kinetic drive has its own CPU and RAM with built-in LevelDB [11]. It can perform the basic

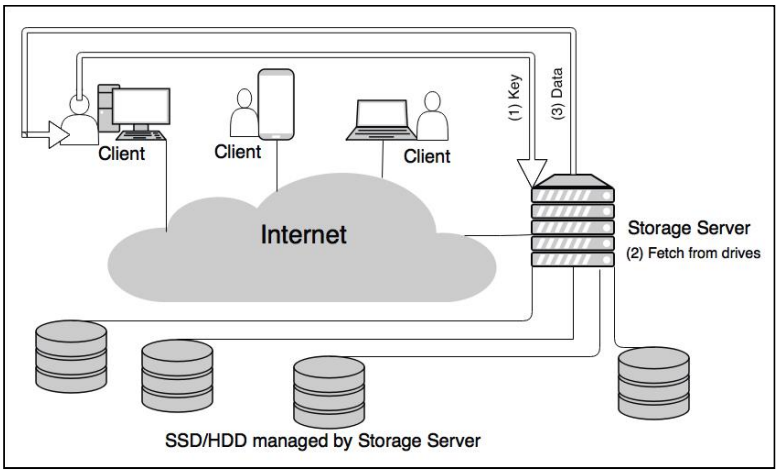
key-value pairs operations efficiently without the need to go through different hardware and software layers introduced by the storage server, shown in Figure 5.1(a). Kinetic drives have an Ethernet connection, shown in Figure 5.1(b). Using this connection, applications can request any data they need by only connecting to the suitable Kinetic drive. Then, data will be transferred over Ethernet to be processed by the applications. So we can say that the Kinetic drive is an independent active device connected to the Internet which can be used as a small key-value store.

In the BigData field and with using hundreds of drives, an allocation schema to place data onto each drive was needed. There is some research done on the Kinetic drive device [34] [33]. The research efforts in [33] proposed a design for the index table based on the metadata server. This metadata server stores some information about the data stored in each Kinetic drive. This index consists of the key range of data stored and the IP address of the drive that holds this key range. However, the proposed approaches in these research efforts never take into consideration the limited bandwidth of the Kinetic drive and the data access frequencies. In this part of our work, we propose a key-value pairs allocation schema. Our scheme takes the limited bandwidth and capacity of the drive as factors in data allocation to satisfy user search requests based on data popularity. To the best of our knowledge, we are the first to explore the data allocation problem with respect to drive bandwidth and data popularity for Kinetic drives.

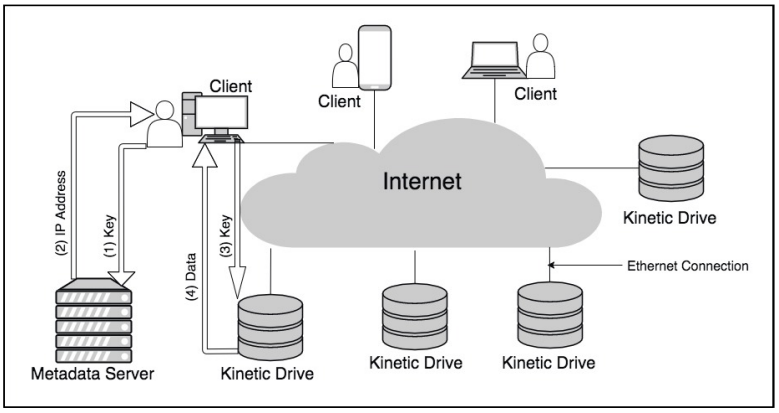
The remaining sections of this chapter are organized as follows. Section 5.2 presents the motivation of the work with kinetic drives. The problem statement including system constraints and assumptions will be presented in Section 5.3. In Section 5.4, a heuristic approach for key ranges placement will be illustrated. In Section 5.5, some experimental results are presented. Section 5.6 presents the related work for data allocation in several environments. Finally, the chapter will be concluded in Section 5.7.

5.2 Motivation and Challenges

The traditional key-value storage implementation is on storage server which manages block storage disk drives. These key-value storages have their own limitations since they consist of multiple layers of software and hardware stacked together in order to enable a



(a) Server-based KV Store System



(b) Kinetic-based KV Store System

Figure 5.2: Comparison between Server-based and Kinetic-based KV Stores

data path between two poorly compatible systems: an object-oriented application layer and a hardware layer (HDDs, SSDs and tape). The transit path from application to storage requires multiple layers of manipulation like databases, POSIX interfaces, file systems, RAID controllers, SAS expanders and SATA host bus adapters [5].

In the traditional server based key-value storage, all requests are sent to the storage server. Then, the storage server handles each request and sends the result back to users as shown in Figure 5.2(a). The previous scenario makes the server-based storage has a performance bottleneck issue. This issue appears when many users share access to a business application, as all requests are sent to the storage server and may build up

in the queue. The response time for each I/O starts to increase, short delays turn into terrible waiting times, which decrease the server throughput and lead to a performance bottleneck [32]. On the side of Kinetic-based key-value storage, shown in Figure 5.2(b), each Kinetic drive works independently as a small storage server. Requests are sent to the metadata server to get the IP address of the drive containing the data. Then, users connect to the drive directly to get the data using the drive's IP address. By following the previous scenario, each Kinetic drive serves only the requests of the data it has. Moreover, in [33], two experiments were conducted to compare the Kinetic-based storage with the server-based storage. The first experiment was conducted to test the write throughput. In this experiment, the Kinetic drive achieves high write throughput in large value size case. The second experiment was to test the data transfer between two Kinetic drives and between two disk drives. The data transfer between two Kinetic drives was faster than the traditional disk drives. It reaches 22.45 MB/s compared to 14.6 MB/s in the case of traditional disk drives.

In the current big data world with enormous amounts of generated data, exploiting parallelism will increase the performance of the storage systems. By taking the advantage of the Kinetic drive as being an independent active device, that can carry out all key-value pair operations on its own, we can exploit parallelism using multiple Kinetic drives instead of using a storage server controlling several HDDs/SDDs. Hence, the server bottleneck is avoided. As a result, the whole storage system performance will be improved. So our goal is to build a Kinetic drive based key-value store. But each Kinetic drive has limited bandwidth and limited capacity. It can only hold data up to its capacity and support data access rate up to its bandwidth. That is why, a careful placement algorithm is needed to resolve the disk bandwidth and capacity issue to avoid queuing at each drive, and hence avoid the performance bottleneck issue on the drive level. On the other hand, the size of the index table, located on the meta-data server, shouldn't be big to speed up the lookup operation for the drives' IP addresses. Having a small index table can also decrease the overhead on the meta-data server to avoid creating another bottleneck point. In this work, we developed a heuristic algorithm to resolve the previous issues. In addition, the algorithm also takes into account decreasing the cost of building the Kinetic-based key-value storage. It minimizes the number of Kinetic drives used to build the key-value storage. By achieving the previous goals,

we will be able to build a low cost Kinetic-based key-value storage, that decreases the response time in satisfying user requests, and increases the overall performance.

5.3 Problem Statement and Assumptions

5.3.1 Problem Statement

Given a set of N identical Kinetic drives, each of size S and bandwidth B, and given a big key range where data is represented with (key, value) pairs, each of size s, in that key range. The distribution of data across that key range is unknown. So the big key range is divided into a set of small key ranges of size M in order to have the data uniformly distributed across each of them. These small key ranges R_1, R_2, ..., R_M also are of different lengths and sizes. Given that users' search requests are not uniformly distributed across all the data, but data in some key ranges are searched by users in high frequencies (hot key ranges) while other data may not be searched frequently like others (cold key ranges), due to the variation in data popularity factor. Hence, the amount of user requests across the drives is unbalanced. Given that the probabilities of accessing data in the small key ranges are p_1, p_2, ..., p_M, respectively, the total bandwidth needed to satisfy all of concurrent access requests for all data in each small key range is H, we want to allocate the (key, value) pairs across the drives using the minimum number of drives to reduce the cost of building the Kinetic drive based KV store. Also we want to reduce the total number of key ranges saved on the index table on the meta-data server to speed up the lookup operation for the drives' IP addresses by combining adjacent key ranges that can fit on one drive into one key range to represent only one record in the index table. For example, if we have three key ranges R_i, R_j, R_k, where $1 \leq i, j, k \leq M$ and R_i, R_j, R_k are consecutive key ranges with n_i, n_j, n_k (key, value) pairs, and access probabilities p_i, p_j, p_k, respectively. These three key ranges can be combined into one key range $Y = [startKey(R_i), endKey(R_k)]$ with n_y key-value pairs, where $n_y = n_i + n_j + n_k$, and p_y access probability, where $p_y = (p_i + p_j + p_k)$, as long as $n_y \times s \leq S$ and $p_y \times H \leq B$.

5.3.2 System Assumptions and Constraints

System Assumptions. We build our problem based on the assumption that the keys across each of the small key ranges are uniformly distributed but the distribution of the data across the whole key range is unknown. Also the access distribution of the data (key-value pairs popularity), bandwidth B and size S of the drive are known in advance. We also assume that the total size of the data is less than or equal to $N \times S$ and the total bandwidth required to access all the data is less than or equal to $N \times B$, i.e. We have a large number of drives that can hold all of our key-value pairs.

Constraints. Total size of the (key, value) pairs in each drive is bounded by the size S of the drive and access distribution of the data stored in each drive is bounded by the bandwidth B of the drive in order that the drive can handle the user requests directed to it without delay. For example, if we have three key ranges R_i, R_j, R_k, where $1 \leq i, j, k \leq M$, with n_i, n_j, n_k (key, value) pairs and access probabilities p_i, p_j, p_k, respectively. The key-value pairs in the three key ranges can be stored on one Kinetic drive with three records in the index table associated with the same drive IP if and only if, conditions (5.1) and (5.2) are true. Also, each Kinetic drive should hold no more than L key ranges, where L is experimentally determined in order not to increase the overhead on the drive of managing multiple key ranges.

$$s * (n_i + n_j + n_k) \leq S \tag{5.1}$$

$$H * (p_i + p_j + p_k) \leq B \tag{5.2}$$

5.4 Heuristic Algorithm to Place Key Ranges in Minimum Number of Drives

Referring to the multi-capacity bin packing problem, we will manipulate each drive as a bin with 3 capacities: size of the drive S, bandwidth of the drive B and number of key ranges that the drive can hold L. Each key range is assigned to a drive if it fits through these 3 capacities. Based on that, we can apply three solutions of the bin packing problem, namely, *Next Fit*, *First Fit*, *Best Fit*. In case of using the Next Fit solution, we will not gain the best results as this algorithm takes the next key range from the

list of key ranges and attempts to place it in the current available Kinetic drive. If this key range doesn't fit within the drive, then a new empty drive is taken from the list of the available empty Kinetic drives to allocate the item to it. So this algorithm doesn't take into consideration that there might be a place for the key range within the list of the non empty drives created before. In case of the First Fit solution, we remove the restriction of the Next Fit by trying to allocate the next key range in the list into any of the currently non-empty drives. This solution chooses the first drive it meets to allocate the key range to it. A new drive will be only picked from the list of the empty drives if the key range couldn't be allocated to any of the non empty ones. In case of the Best Fit solution, it behaves similar to the First Fit solution, however, it is different in the criteria of how to choose the best candidate drive to place the key range into it. This solution doesn't place the item in the first drive it meets, that could fit the item, but it considers all the drives and chooses the one which will have the least empty space after placing the item on it. If we directly apply the Best Fit solution, then we could have unbalanced capacities left in the drive. This may result in consuming one capacity and leaving the others nearly untouched. So we proposed a heuristic algorithm to allocate the key ranges into minimum number of Kinetic drives with taking into consideration the balance among the used size, bandwidth and number of key ranges that the drive can support and also with minimizing the number of records in the index table saved on the meta-data server. Our algorithm consists of three steps carried out sequentially: *Preprocessing of Key Ranges*, *Sorting of Key Ranges* and *Key Ranges Allocation*. We will discuss each step in details throughout this section.

5.4.1 Preprocessing of Key Ranges

In order to decrease the number of key ranges in the index table for each drive, we will perform a kind of preprocessing to combine some consecutive key ranges into one key range with probability of access equal to the sum of the combined key ranges probabilities and number of key-value pairs equal to the sum of combined key ranges key-value pairs. The preprocessing step is carried out on two phases: combining based on access profile and combining based on drive specification as shown in Algorithm 2. In the phase of combining based on access profile, starting from line 4, the following steps are carried out:

- Sorting the key ranges in ascending order according to their starting point.
- Iterating over all key ranges, starting from the first one, and packing multiple key ranges into one key range if they have close access profile and can fit into one drive.
- If the drive threshold (bandwidth or size) is met, we start another packing step over the remaining ranges till we process all key ranges.

If we have 3 consecutive key ranges R_i, R_j, R_k, where $1 \leq i, j, k \leq M$, with n_i, n_j, n_k (key, value) pairs and access probabilities p_i, p_j, p_k, respectively, the 3 key ranges can be combined if (5.1), (5.2), and (5.3) are true.

$$p_i \approx p_j \approx p_k \tag{5.3}$$

In the phase of combining based on the drive specification, starting from line 17, we begin another combining step, based on the drive threshold values only (bandwidth and size), by iterating over all key ranges, starting from the first one, and packing multiple key ranges into one key range if they can fit into one drive, i.e., (5.1) and (5.2) are true.

5.4.2 Effect of Sorting of Key Ranges according to a Weighted Function

There could be different variations of the best fit algorithm based on the different ordering of the key ranges. However, the ordering should be based on the bandwidth requirement and the storage requirement of the key ranges. We sort the data in a descending order according to the weighted function W which is defined in [85] as a combination of the storage requirement for an item X as s_x and the bandwidth requirement b_x as follows:

$$W = (\frac{N_{access}^{min}}{N_{storage}^{min}} \times b_x) + \beta \times (\frac{N_{storage}^{min}}{N_{access}^{min}} \times s_x) \tag{5.4}$$

$$N_{access}^{min} = \frac{H \times \sum_{i=1}^{M} p_i}{B}, N_{storage}^{min} = \frac{s \times \sum_{i=1}^{M} n_i}{S} \tag{5.5}$$

where N_{access}^{min} is the minimum number of drives needed to satisfy the total bandwidth requirements for all M key ranges and $N_{storage}^{min}$ is the minimum number of drives needed to store the n_i key-value pairs, each of size s, for the M key ranges, where $1 \leq i \leq M$. β is a positive integer used to balance the storage load with the concurrent access load so when treating both loads equally, β should be set to be 1. The cost function W is adaptive to the key ranges load as it favors the bandwidth requirement if the key ranges bandwidth requirement is the critical factor, i.e. $N_{access}^{min} > N_{storage}^{min}$, and favors the storage requirements if the key ranges storage is the critical factor, i.e. $N_{storage}^{min} > N_{access}^{min}$

Algorithm 2 Key Ranges Pre-processing

Input: key_ranges: List of key ranges.
Result: key_ranges: Combined Key ranges .
Begin:
Sort the list of key ranges in ascending order according to starting key.
for All key ranges in the list **do**
  if $s_i + s_{i+1} \leq S$ **and** $b_i + b_{i+1} \leq B$ **then**
    if $(b_i - b_{i+1}) \leq \delta$ **then**
      combine the key ranges i and $i+1$
      continue to the whole list starting from the combined key range.
    else
      continue to the whole list starting from key range $i+1$
    end if
  else
    continue to the whole list starting from key range $i+1$
  end if
end for
for All key ranges in the list **do**
  if $s_i + s_{i+1} \leq S$ **and** $b_i + b_{i+1} \leq B$ **then**
    combine the key ranges i and $i+1$
    continue to the whole list starting from the combined key range.
  else
    continue to the whole list starting from key range $i+1$
  end if
end for

5.4.3 Allocation Using Variation of Best Fit Algorithm

To avoid the limitations of the three solutions mentioned above at the beginning of this section, we propose a best fit-like algorithm, shown in Algorithm 3, which selects the best candidate drive to allocate the key range based on the following criteria.

Choosing the best candidate criteria. If we have N non empty drives each of them has $b_i\%$ used bandwidth of the total bandwidth B and $s_i\%$ used space from the total space S and L_i allocated key ranges, where $1 \leq i \leq N$, we want to choose one of them as a candidate to allocate the key range X which has $b_x\%$ bandwidth and $s_x\%$ size requirements, the best candidate is the drive that minimizes the following equation:

$$|(b_i + b_x) - (s_i + s_x)| \tag{5.6}$$

Algorithm 3 Key Ranges Allocation

```
1: Input: key_ranges : List of key ranges.
2: Result: List of drives with allocated key ranges .
3: Begin:
4: for all key ranges in the list do
5:   if a candidate non-empty drive is found then
6:     Add the key range to the index table associated with drive IP
7:     Update the total bandwidth and size used for the drive
8:     Continue to the following key range.
9:   else
10:    if lower bound on number of drives reached then
11:      if length(Key range) ≤ minimum length then
12:        Add new empty drive to the system.
13:        Allocate the key range to the drive.
14:        Continue to the following key range.
15:      else
16:        Divide the key range according to the maximum empty slot in used drives
17:      end if
18:    else
19:      Allocate the key range to one empty drive.
20:      Update the new drive's total used size and total used bandwidth.
21:      Continue to the following key range.
22:    end if
23:  end if
24: end for
```

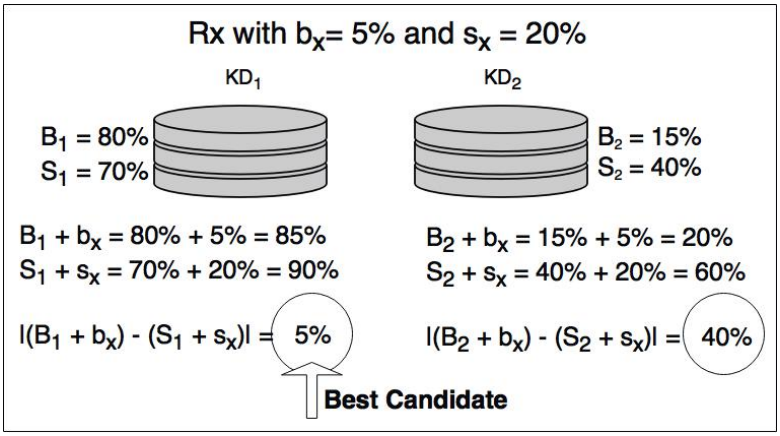


Figure 5.3: Example of the best candidate choice

Where $b_i + b_x < 100$, $s_i + s_x < 100$ and $L_i + 1 \leq L$, i.e the key range can fit into the drive. Equation 5.6 ensures the balance between the drive used size and used bandwidth. Figure 5.3 shows an example of two Kinetic drives, each of them can be a candidate to place the key-range X into it. But according to the best candidate criteria we discussed in Equation 5.6, the Kinetic drive KD_1 is the best candidate for X. If the used bandwidth percentage or the used size percentage of the drive will be $\leq 70\%$ after adding the key range X, while the number of key ranges on the drive will be equal to the maximum number of key ranges/drive L, then this drive will be removed from the best candidate list because the drive will not be able to hold more key ranges despite of having unused bandwidth and unused size that could make it the best candidate for other key ranges. If there is no non-empty drive found as a candidate to allocate the current key range, then one of the following cases will be applied:

- If the lower bound on number of drives that needed to allocate all key ranges were consumed without allocating all the key ranges, then the current key range is divided into two parts: The first part equals to the maximum empty slot on the non-empty drives, and the second part is the rest of the key range. The first part is allocated on the drive with the maximum empty slot and the second part is added to the key ranges list to continue the allocation through the algorithm. If the minimum length of the key range is reached then we add new drive to the system to allocate the key range into it.

- Otherwise, a new empty drive is added to the system to hold the key range.

When the key range needs to be divided into two parts for allocation, then we have two cases. The first case is when the access requirement is uniformly distributed across the key range and the key-value pairs are uniformly distributed across the key range. In this case, the key range is divided into two parts according to the empty slot available. The second case is when the distribution of the access requirement is unknown. In this case, the key range is fully replicated with dividing the access requirement between the two replicas according to the empty slot found on the non empty drive.

5.5 Performance Evaluation

5.5.1 Experiment Setup

In this section, we experimentally evaluate the performance of our key-value pairs allocation approach "*Merge-Divide*". We compare our proposed approach with other two variants of allocation approaches and the base-line numbers determined theoretically. In our Comparison, we used different workloads and different distributions for the access request and key-value distribution. The first variant called "*NoMerge-NoDivide*" is allocating the key ranges onto the drives without combining phase at the beginning and also without dividing the key ranges if the optimal number of drives is reached. The key ranges are allocated on the drives according to the best fit bin packing algorithm using our best candidate criteria defined in 5.4.3. The second variant called "*Merge-NoDivide*" is to combine the consecutive key ranges at the beginning but without dividing again when the minimum number of drives is consumed. To the best of our knowledge, our allocation approach is the first allocation approach specifically designed for kinetic drives that takes into consideration the limited access request and the limited size of the drive. The closest competitor to us is the data management approaches for kinetic drives defined in [33], they design 4 approaches to maintain the index table but without taking into consideration the limited bandwidth of the kinetic drive and the data access frequencies. They also didn't take into consideration how to minimize the index table to decrease the look up time on the meta-data server.

Workloads. We generate a large number of key-value pairs, these key-value pairs are divided into a set of key ranges, each key-range has a size and access request requirements, the access request requirements are generated using the zipf distribution generator of Yahoo! Cloud Serving Benchmark (YCSB) [86]. For the key-value pair generation, we generate key-value pairs in different key distributions (zipf and uniform) using the YCSB generator. Table 5.1 shows the different workloads and the distribution of key-value pairs and access requests.

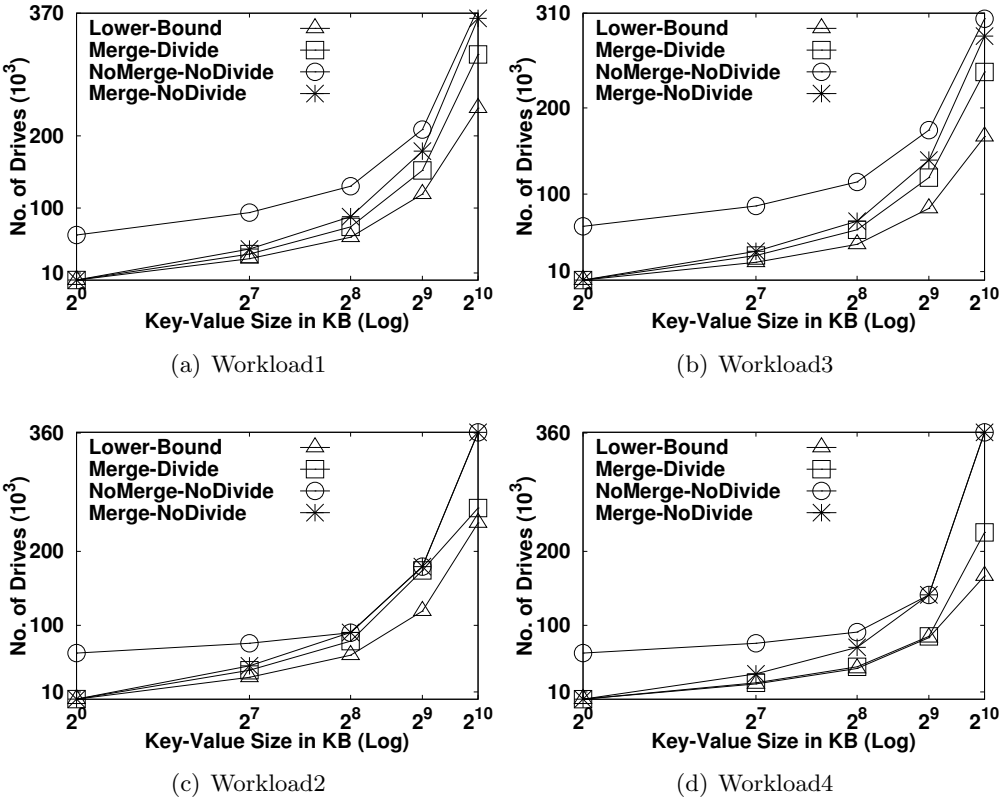
Workload	No. Key-Value pairs	No. Concurrent Access Requests	Key-Value Distribution	Access Distribution
Workload1	1 Trillion	1 Million	Zipf Distribution	Zipf Distribution
Workload2	1 Trillion	1 Million	Uniform Distribution	Zipf Distribution
Workload3	6 Billions	1 Million	Zipf Distribution	Zipf Distribution
Workload4	6 Billions	1 Million	Uniform Distribution	Zipf Distribution

Table 5.1: Key-Value pair size and its associated maximum key-value pairs/drive and maximum access requests/drive

Parameters. The current available model of the kinetic drive ST4000NK0001 [4] has a storage capacity of 4 TB and can support a transfer rate up to 60 MB/s. The maximum Key-value pair size supported by the drive is 1 MB. We use this previous configuration in our simulation steps and we varied the key-value pair size in our simulation that the drive can support to test our algorithm under different key-value pair sizes. The key-value pair size with its associated maximum key-value pairs/drive, and maximum requests/second that the drive can sustain are shown in Table 5.2. We assume that we have enough number of drives that can hold all the key-value pairs. We also assume that the maximum number for key ranges/drive is 5 key ranges, and the minimum access request/key range that we can't divide the key range again when reaching it is 4. Also we set the minimum number of key-value pairs/key range to be 500000 key-value pair. These parameter values are chosen in order to achieve good reduction ration in the index table size with minimizing also the total number of drives.

Key-value size	1KB	128KB	256KB	512KB	1024KB
maximum kv/drive	4294967296	33554432	16777216	8388608	4194304
maximum access/drive	61440	480	240	120	60

Table 5.2: Key-Value pair size and its associated maximum key-value pairs/drive and maximum access requests/drive



(a) Workload1 (b) Workload3

(c) Workload2 (d) Workload4

Figure 5.4: Effect on Number of Drives Using Different Distribution for Key-Value Pairs and Access Requests

Metrics. In all experiments, we use the total number of drives used to allocate the key ranges and the size of the index table on the meta-data server as our performance metrics.

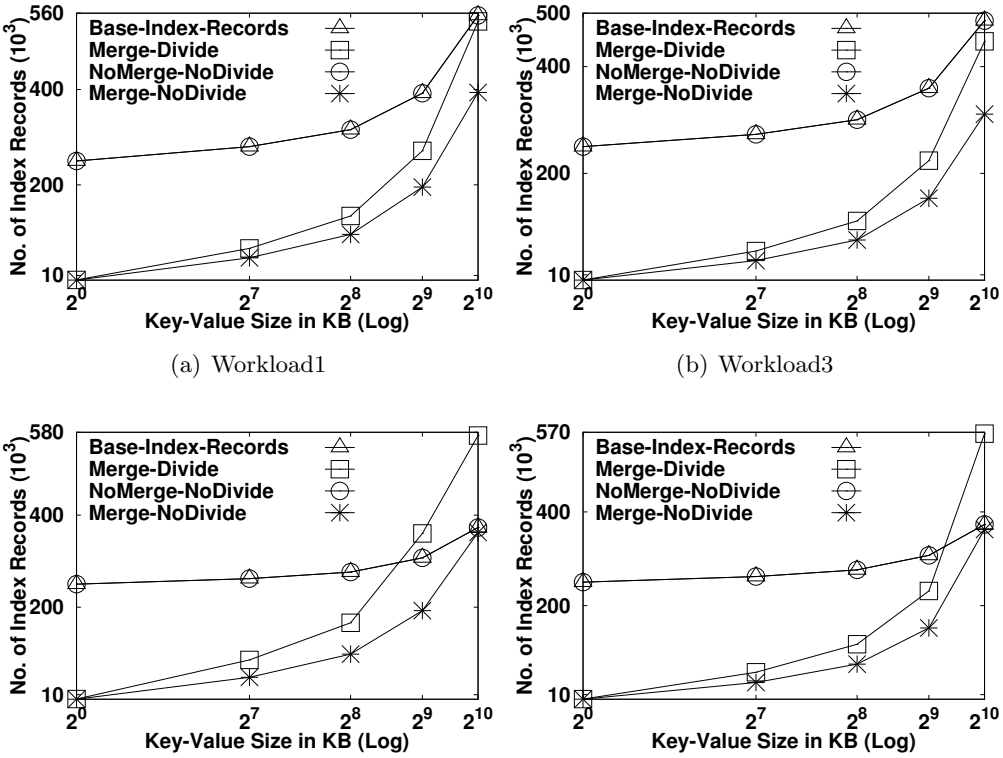
5.5.2 Effect on Number of Drives Used

In this section, we compare the performance of our approach "*Merge-Divide*" with the other two variants; "*NoMerge-NoDivide*" and "*Merge-NoDivide*", on the total number of drives used and the base-line lower bound of number of drives determined theoretically from $max\{N^{min}_{access}, N^{min}_{storage}\}$, where N^{min}_{access} and $N^{min}_{storage}$ can be determined from Equation 5.5 using the same workloads defined in Table 5.1 and the same configuration parameters shown in Table 5.2.

Using Zipf key and Zipf access distributions. In this case, we use Zipf distribution for the key-value distribution to test our approach with skewed data, as some key ranges may be dense that has large number of key-value pairs, and other may be scarce that has small number of key-value pairs. We also used the Zipf distribution for the access request which is typically used when modeling data popularity, because data usually differs in its access popularity, some data is hot and the rest of the data is cold.

Figure 5.4(a) shows the results of the number of drives used in the 4 cases using workload1 where the total size of the data is the critical factor and the lower bound is determined by the minimum number of drives to satisfy the size requirement. It shows that our approach "*Merge-Divide*" outperforms the "*Merge-NoDivide*" with average of 38459 fewer drives in large key-value pair size ($512KB, 1024KB$) and with average of 7071 fewer drives in small key-value pair sizes ($1KB, 128KB, 256KB$). This result is because we divide some of the small key ranges to fill empty gaps in the non empty drives instead of using new drives to allocate them. we usually fill the biggest gaps first in order to minimize the number of divisions and hence minimize the index table. We found also that our approach outperforms the "*NoMerge-NoDivide*" in all cases of key-value pair sizes with average of 59349 drives, this is because the drives reach their capacity of the number of key ranges/drive before filling all other capacities, because of the large number of key ranges that we have without having the merging phase. Figure 5.4(a) also shows that our approach is the closest one to the base-line lower bound with 25588.8 drives above the lower bound on average compared to 45000+ drives in the other 2 approaches.

Figure 5.4(b) shows the results of the number of drives used in the 4 cases using workload3 where the total concurrent access requests of the data is the critical factor and the lower bound is determined by the minimum number of drives to satisfy the access request requirement. It shows that our approach "*Merge-Divide*" outperforms the "*Merge-NoDivide*" with average of 15352 fewer drives. We found also that our approach outperforms the "*NoMerge-NoDivide*" with average of 58436 drives and our approach is the closest one to the base-line lower bound on average. This is because we adjust the sorting phase of our algorithm according to the critical factor to allocate the largest key ranges that need more space of the critical factor first, and hence get better allocation results.



(a) Workload1 (b) Workload3

(c) Workload2 (d) Workload4

Figure 5.5: Effect on The Size of Index Table Using Different Distribution for Key-Value Pairs and Access Requests

Using uniform key and Zipf access distributions. In this case, we use uniform distribution for the key-value pairs to test our approach under different key-value pairs distribution, while keeping the access request generated by Zipf distribution as a typical case in most storage access patterns. We tested our 4 cases using workload2 where the size requirement is the critical factor and workload4 where the concurrent access request requirement is the critical factor.

Figure 5.4(c) and Figure 5.4(d) show the results of the number of drives used in the 4 cases using workload2 and workload4 respectively, where the lower bound is determined by the minimum number of drives to satisfy the critical factor requirement. It shows that our approach "*Merge-Divide*" outperforms the other two variants and is the closest one to the base-line lower bound. We have also tested our approach for other workloads

where the access request is uniform on all key ranges which is not the typical case in storage access patterns and it shows that it performs well in other distributions and it is the closest one to the lower bound.

5.5.3 Effect on The Size of Index Table

In this section, we compare the performance of our approach "*Merge-Divide*" with the other two variants; "*NoMerge-NoDivide*" and "*Merge-NoDivide*", on the total size of the index table resulted after allocation and the starting size of the index table before the allocation to determine the reduction in the size of the index table using the same workloads defined in Table 5.1 and the same configurations shown in Table 5.2.

Figure 5.5(a) and Figure 5.5(b) show that reduction in the index table size in case of "*Merge-NoDivide*" approach is higher than the reduction in case of "*Merge-Divide*" on average, where their reduction ratios for workload1 are 66.4% and 53.3%, and their reduction ratios in workload3 are 70.9% and 57.7%, respectively. This results because we don't divide the key ranges again in "*Merge-NoDivide*" so we get the highest reduction result. The two figures also show that the reduction in the index table in case of small key-value pair size is higher than the reduction in case of large key-value pairs size. This result is because in small key-value pair sizes the number of key-value pairs and the number of concurrent access requests that the drive can support are larger than these numbers in case of large key-value pair sizes. This allows merging large number of key ranges into one key range and hence reduces the size of the index table records. On the other hand, the "NoMerge-NoDivide" approach doesn't have any reduction performance in the index table size because it doesn't have merging phase at the beginning.

Figure 5.5(c) and Figure 5.5(d) show that reduction in the index table size in case of "*Merge-NoDivide*" approach is also higher than the reduction in case of "*Merge-Divide*" on average. Their reduction ratios are 57.3% and 27.2% in case of workload2, and 62.5% and 41.6% in case of workload4, respectively. These results are because in large key-value pair size, the "*Merge-Divide*" divides the key ranges several times before reaching the minimum length of the key range to fill the gabs on the non empty drives to achieve better performance in minimizing the number of drives used. But on average of all key-value pair sizes, the "*Merge-Divide*" still achieves good reduction ratio.

To sum up, the"*Merge-Divide*" achieves a balance between the reduction in the

number of drives used and the reduction of the index table size to allocate key-value pairs on the kinetic drives, while taking into consideration the limited size and bandwidth of the drives.

5.6 Related Work

The data allocation problem was studied in different fields, to the best of our knowledge, we are the first to explore this problem with respect to bandwidth on the kinetic drives. There are a lot of problems related to data allocation which we will discuss in this section. We will show some related problems and work done in each one of them, then the differences between related problems and our problem will be conducted.

One of the related problem is data allocation problem for Video On Demand (VOD) on multimedia servers. In this problem, depending on the popularity of each movie, a file placement algorithm was needed to allocate the movie files over a number of magnetic disk arrays such that a given access profile can be supported at a minimum delay cost. In [85], the previous problem was proved to be NP problem and they proposed a heuristic gready approach to place the movies on minimum number of disk arrays and while considering file replication and stripping in their approach. In [87], They formulated the problem as a transportation problem and solved the problem to find a near optimal solution.

The second problem is the data allocation for multiple wireless broadcast channels. In this problem, the broadcast bandwidth is a scarce resource in the mobile computing environment. Hence, a lot of work done in [88] [89] [90] [91] to efficiently allocate data on the broadcast channel to speed up the data access for users. In [88], They formulated the problem as the famous knapsack problem and developed a gready approach to solve the problem. In [89], They developed a data allocation schema based on replicating the data items on the wireless channel to satisfy the access requirements from user queries so users will not wait for the second cycle to access the data. In their approach, number of replicas of a data item is determined according to the relative frequency of access between the data item and the least frequent data item in the system to minimize the overall access latency of the system. In [90], a two-stage scheme of initial placement generation and placement refinement was developed to solve the placement

problem. In [91], A heuristic algorithm was proposed to select a range of data items to be allocated on the broadcast channel. The selected range is the one which achieves the Minimal Expected Average Access Time (MEAAT) according to a predefined objective function.

The third problem is the file allocatin problem on disks managed by severs in order to achieve load balancing across these servers. There is a lot of work done in this problem as shown in [92] [93] [94] [95].

The previous problems differ from our problem. In our problem, we are allocating key ranges to each drive according to size and access requirement of each key range. We also take into consideration the design of the meta-data server index table; we try to minimize the number of records in the index table to minimize the searching time in the index table to get the IP of the kinetic drive and not to have bottleneck issue on the meta-data server itself.

5.7 Conclusion

In the BigData world with its huge amount of data, there was indeed a need for tools to be able to store and process this huge amount of data efficiently. One of these tools is the key-value store. With the invention of the kinetic drive and being used as an independent key-value store, we can exploit parallelism to increase the performance of the whole storage system. In this paper, we proposed a data allocation approach to allocate key-value pairs on the kinetic drives taking into consideration the bandwidth issue of the kinetic drive to avoid queuing on the level of the drive. We also took the size of the index table into consideration to speed up the search time for the drive's IP in the meta-data server. The performance evaluation shows that our proposed approach is near-optimal in terms of minimizing the number of assigned drives.

Chapter 6

Conclusion

Big Data has drawn lots of people's attention nowadays. Enormous amount of data has been generated and analyzed for the benefit of society at a large scale. With this huge amount of generated data, data is distributed among several storage instances, accessed frequently, retrieved and processed by many applications to extract useful information. So, it is important to improve the data access performance when data is accessed from storage nodes through network. For the nature of nowadays data, it is usually maintained in key-value stores. In our work, we focused on how to improve data access performance from key-value stores using some hardware accelerators on the network side or on the storage side.

In Chapter 2, we focused on the preliminaries of the emerging hardware technologies we used to enhance the performance of distributed key-value stores. We illustrated the main components inside the programmable network switches and what is the processing flow inside them. We also gave a brief summary about the Object storage devices like kinetic drives and their functionalities and specifications.

In Chapter 3, we presented *TurboKV*; a novel distributed key-value store architecture that leverages the power and flexibility of the new programmable switches. *TurboKV* uses the in-switch coordination approach that utilizes the switches as partitions management nodes to store the key-value store partitions locations and replicas information along the path from clients to storage nodes. The programmable switches use key-based routing to route packets from clients to storage nodes. We believe that *TurboKV* can be deployed on the programmable switches currently integrated in the

data center's network to improve the performance of distributed key-value stores.

In Chapter 4, we presented an overview of our proposed networking support for transaction processing using programmable switches. Our proposed approach utilizes the programmable switches to execute the transactions processing logic in the network. If a transaction can be pushed to processing by acquiring all of its needed data, it is accepted and forwarded to the target storage nodes to start processing. Otherwise, the transaction is aborted early by the programmable switches. We believe that this approach will have significant impact on the performance of transactions network latency has a significant impact on the performance of transactions which have to be processed by the storage system in order to ensure serializability. We implement a variation of the Time Stamp Ordering algorithm (TSO) on the programmable switches. by re-writing the packet header, and routing the packets back to the client.

In Chapter 5, we presented a heuristic algorithm to allocate key-value pairs on the kinetic drives taking into consideration the bandwidth issue of the kinetic drive to avoid queuing on the level of the drive. We also the size of the index table, stored on the meta-data server, into consideration to speed up the search time for the drive's IP address. In addition, our proposed algorithm takes into account decreasing the cost of building the kinetic-based key-value storage. It minimizes the number of kinetic drives used to build the key-value storage.

References

[1] Hebatalla Eldakiky and David H.C. Du. Turbokv: Scaling up the performance of distributed key-value stores with in-switch coordination. In *2020 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Under Review 2021.

[2] Hebatalla Eldakiky and David H.C. Du. Transkv: A networking support for transaction processing in distributed key-value stores. In *2021 IEEE Seventh International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 17–24, 2021.

[3] Hebatalla Eldakiky and David H.C. Du. Key-value pairs allocation strategy for kinetic drives. In *2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 17–24, 2018.

[4] "Seagate® Kinetic HDD Standard model ST4000NK0001 product Manual", https://www.seagate.com/www-content/product-content/hdd-fam/kinetic-hdd/_shared/docs/kinetic-product-manual.pdf.

[5] "Seagate Kinetic Storage vision": http://www.seagate.com/tech-insights/kinetic-vision-how-seagate-newdeveloper-tools-meets-the-needs-of-cloud-storage-platforms-master-ti/.

[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, pages 2787–2805, 10 2010.

[7] "EMC Digital Universe with Research & Analysis by IDC.": https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07. ACM, 2007.

[9] S. Sanfilippo and P. Noordhuis, Redis: in-memory Key-value store, http://redis.io, 2009.

[10] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4), January 2010.

[11] LevelDB: A light-weight single-purpose library for persistent key-value store, https://github.com/google/leveldb.

[12] RocksDB. A facebook fork of leveldb which is optimized for flash and big memory machines, 2013. https://rocksdb.org.

[13] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38, 2008.

[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44, 2014.

[15] P4Runtime: https://p4.org/p4-runtime/.

[16] Google Cloud using P4Runtime to build smart networks: https://cloud.google.com/blog/products/gcp/google-cloud-using-p4runtime-to-build-smart-networks.

[17] ATT usage of Programmable switches: https://www.sdxcentral.com/articles/news/att-picks-barefoot-networks-programmable-switches/2017/04/.

[18] Barefoot Networks: https://www.barefootnetworks.com/ press-releases/barefoot-networks-unveils-tofino-2-the-next-generation-of-the-worlds-first-fully-p4-programmable-network-switch-asics/.

[19] Brent Stephens, Aditya Akella, and Michael M. Swift. Your programmable nic should be a programmable switch. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18. ACM, 2018.

[20] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, New York, NY, USA, 2015. ACM.

[21] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2):18–24, May 2016.

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17. ACM, 2017.

[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, 2018. USENIX Association.

[24] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File*

and Storage Technologies (FAST 19), pages 143–157, Boston, MA, 2019. USENIX Association.

[25] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 279–291, New York, NY, USA, 2019. ACM.

[26] "Intel Tofino.": https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[27] "Intel Tofino Second Generation.": https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[29] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010.

[30] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: Architecture of a real-time operational dbms. *Proc. VLDB Endow.*, 9(13):1389–1400, September 2016.

[31] David P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23, February 1983.

[32] "I/O botteleneck of storage servers": http://searchstorage.techtarget.com/report/Troubleshooting-andidentifying-data-storage-performance-bottlenecks.

[33] X. Cao, M. Minglani, and D. H. Du. Data allocation of large-scale key-value store system using kinetic drives. In *2017 IEEE Third International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 60–69, April 2017.

[34] M. Minglani, J. Diehl, X. Cao, B. Li, D. Park, D. J. Lilja, and D. H. C. Du. Kinetic action: Performance analysis of integrated key-value storage devices vs. leveldb servers. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 501–510, Dec 2017.

[35] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 387–406. USENIX Association, November 2020.

[36] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292. USENIX Association, February 2021.

[37] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 40(1), June 2012.

[38] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10. ACM, 2010.

[39] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[40] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44, 2014.

[41] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38, 2008.

[42] P4 Software Switch Website: https://github.com/p4lang/behavioral-model.

[43] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 203–216, Savannah, GA, 2016. USENIX Association.

[44] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. Ripemd-160: A strengthened version of ripemd. In *Proceedings of the Third International Workshop on Fast Software Encryption*, pages 71–82, London, UK, UK, 1996. Springer-Verlag.

[45] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[46] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Distributed systems (2nd ed.). chapter The Primary-backup Approach, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[47] Mininet Website: http://mininet.org/.

[48] Scapy for Packet Manipulation Website: https://scapy.readthedocs.io/en/latest/.

[49] Python Interface for LevelDB: https://plyvel.readthedocs.io/en/latest/.

[50] Pegasus BMV2 Repository: https://github.com/NUS-Systems-Lab/pegasus/tree/master/p4/bmv2.

[51] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, 2013.

[52] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with switchkv. In *Proceedings of the 13th*

Usenix Conference on Networked Systems Design and Implementation, NSDI'16, pages 31–44, Berkeley, CA, USA, 2016. USENIX Association.

[53] Hao Wen, Zhichao Cao, Yang Zhang, Xiang Cao, Ziqi Fan, Doug Voigt, and David Du. Joins: Meeting latency slo with integrated control for networked storage. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 194–200. IEEE, 2018.

[54] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[55] L. Woods, Z. István, and G. Alonso. Ibex-an intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment*, 7:963–974, 01 2014.

[56] Database Ranking categories listed by DB-engine: https://db-engines.com/en/ranking_categories.

[57] Customers of Popular Key-value stores listed by DB-engine: https://db-engines.com/en/ranking/key-value+store.

[58] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[59] Amazon DynamoDB support for transaction processing: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/transactions.html.

[60] H. Che, Z. Wang, and Y. Tung. Analysis and design of hierarchical web caching systems. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 3, pages 1416–1424 vol.3, 2001.

[61] Zhen Liu, Philippe Nain, Nicolas Niclausse, and Don Towsley. Static caching of Web servers. In Kevin Jeffay, Dilip D. Kandlur, and Timothy Roscoe, editors, *Multimedia Computing and Networking 1998*, volume 3310, pages 179 – 190. International Society for Optics and Photonics, SPIE, 1997.

[62] Eman Ramadan, Arvind Narayanan, Zhi-Li Zhang, Runhui Li, and Gong Zhang. Big cache abstraction for cache networks. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 742–752, 2017.

[63] Michele Garetto, Emilio Leonardi, and Valentina Martina. A unified approach to the performance analysis of caching systems. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 1(3), May 2016.

[64] Pariya Babaie, Eman Ramadan, and Zhi-Li Zhang. Cache network management using big cache abstraction. 04 2019.

[65] David Starobinski and David Tse. Probabilistic methods for web caching. *Perform. Eval.*, 46(2–3):125–137, October 2001.

[66] Ramez Elmasri et al. *Fundamentals of Database Systems, 3rd Edition*. Addison-Wesley-Longman, 2000.

[67] George Karakostas and Dimitrios Serpanos. Practical lfu implementation for web caching. 06 2000.

[68] shah Ketan, Mitra Anirban, and Matani Dhruv. An o(1) algorithm for implementing the lfu cache eviction scheme. 07 2021.

[69] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[70] MongoDB. https://www.mongodb.com/.

[71] CouchDB. https://couchdb.apache.org/.

[72] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 2653–2666, 2021.

[73] Edward Bortnikov, Eshcar Hillel, Idit Keidar, Ivan Kelly, Matthieu Morel, Sameer Paranjpye, Francisco Perez-Sorrosal, and Ohad Shacham. Omid, reloaded: Scalable and highly-available transaction processing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 167–180, Santa Clara, CA, February 2017. USENIX Association.

[74] Ohad Shacham, Yonatan Gottesman, Aran Bergman, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Taking omid to the clouds: Fast, scalable transactions for real-time cloud analytics. *Proc. VLDB Endow.*, 11(12):1795–1808, August 2018.

[75] Akon Dey, Alan Fekete, and Uwe Röhm. Scalable transactions across heterogeneous nosql key-value data stores. *Proc. VLDB Endow.*, 6(12):1434–1439, August 2013.

[76] Akon Dey, Alan Fekete, and Uwe Röhm. Scalable distributed transactions across heterogeneous stores. volume 2015, April 2015.

[77] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Lightweight Multi-Key Transactions for Key-Value Stores. *CoRR*, abs/1509.07815, 2015.

[78] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 9:87094–87155, 2021.

[79] Theo Jepsen, Leandro Pacheco de Sousa, Huynh Tu Dang, Fernando Pedone, and Robert Soulé. Gotthard: Network Support for Transaction Processing. In *Proceedings of the Symposium on SDN Research*, SOSR '17, page 185–186, 2017.

[80] "NoSQL Database Lectures": http://www.christof-strauch.de/nosqldbs.pdf.

[81] Ken north, "databases in the cloud", article in dr. drobb's magazine, 2009.

[82] Bob ippolito, "drop acid and think about data", talk at pycon, 2009.

[83] NoSQL Databases Website: http://nosql-database.org/.

[84] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12. USENIX Association, 2012.

[85] Yuewei Wang, Jonathan C. L. Liu, David H. C. Du, and Jenwei Hsieh. Efficient video file allocation schemes for video-on-demand services. *Multimedia Syst.*, 5(5):283–296, September 1997.

[86] "YCSB Github Repository": https://github.com/brianfrankcooper/YCSB/wiki.

[87] Kairui Chen, Hui-Chuan Chen, Richard Borie, and Jonathan C. L. Liu. File Replication in Video on Demand Services. In *Proceedings of the 43rd Annual Southeast Regional Conference - Volume 1*, page 162–167, 2005.

[88] Guanling Lee and Shou-Chih Lo. Broadcast data allocation for efficient access of multiple data items in mobile environments. *MONET*, 8:365–375, 08 2003.

[89] Sung-wook Park and Sungwon Jung. A data allocation scheme for multiple wireless broadcast channels. In *Proceedings of the 2nd International Conference on Ubiquitous Information Management and Communication*, ICUIMC '08, page 256–262, 2008.

[90] Guang-Ming Wu. An efficient data placement for query-set-based broadcasting in mobile environments. *Comput. Commun.*, 30(5):1075–1081, March 2007.

[91] Chih-Hao Hsu, Guanling Lee, and Arbee L. Chen. An efficient algorithm for near optimal data allocation on multiple broadcast channels. *Distrib. Parallel Databases*, 18(3):207–222, 2005.

[92] Amjad Mahmood, H. Khan, and H. Fatmi. Adaptive file allocation in distributed computer systems. *Distributed Systems Engineering*, 1:354–361, 12 1994.

[93] Wen-Chih Peng and Ming-Syan Chen. Developing data allocation schemes by incremental mining of user moving patterns in a mobile computing system. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):70–85, 2003.

[94] N. Daudpota. Five steps to construct a model of data allocation for distributed database systems. *Journal of Intelligent Information Systems*, 11:153–168, 2004.

[95] K. M. Chandy and J. E. Hewes. File allocation in distributed systems. In *Proceedings of the 1976 ACM SIGMETRICS Conference on Computer Performance Modeling Measurement and Evaluation*, SIGMETRICS '76, page 10–13, 1976.